# An implementation research on software defect prediction using machine learning techniques

Laur Pulliainen

Helsinki September 10, 2018

Master's thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

| Tiedekunta — Fakultet — Faculty | Laitos — Institution — Department |
|---|---|
| Faculty of Science | Department of Computer Science |

Tekijä — Författare — Author
Laur Pulliainen

Työn nimi — Arbetets titel — Title

An implementation research on software defect prediction using machine learning techniques

Oppiaine — Läroämne — Subject
Computer Science

| Työn laji — Arbetets art — Level | Aika — Datum — Month and year | Sivumäärä — Sidoantal — Number of pages |
|---|---|---|
| Master's thesis | September 10, 2018 | 58 pages + 3 appendix pages |

Tiivistelmä — Referat — Abstract

Software defect prediction is the process of improving software testing process by identifying defects in the software. It is accomplished by using supervised machine learning with software metrics and defect data as variables. While the theory behind software defect prediction has been validated in previous studies, it has not widely been implemented into practice. In this thesis, a software defect prediction framework is implemented for improving testing process resource allocation and software release time optimization at RELEX Solutions. For this purpose, code and change metrics are collected from RELEX software. The used metrics are selected with the criteria of their frequency of usage in other software defect prediction studies, and availability of the metric in metric collection tools. In addition to metric data, defect data is collected from issue tracker. Then, a framework for classifying the collected data is implemented and experimented on. The framework leverages existing machine learning algorithm libraries to provide classification functionality, using classifiers which are found to perform well in similar software defect prediction experiments. The results from classification are validated utilizing commonly used classifier performance metrics, in addition to which the suitability of the predictions is verified from a use case point of view. It is found that software defect prediction does work in practice, with the implementation achieving comparable results to other similar studies when measuring by classifier performance metrics. When validating against the defined use cases, the performance is found acceptable, however the performance varies between different data sets. It is thus concluded that while results are tentatively positive, further monitoring with future software versions is needed to verify performance and reliability of the framework.

ACM Computing Classification System (CCS):
**Software and its engineering → Software defect analysis**
**Computing methodologies → Supervised learning by classification**
Computing methodologies → Cost-sensitive learning
Computing methodologies → Ensemble methods

Avainsanat — Nyckelord — Keywords
software defect prediction, machine learning, supervised learning, software metrics

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

# Contents

# 1 Introduction

Software defect prediction is the process of using software metrics to predict defective components in a software. Software defect prediction is associated with several benefits [1]. It complements software testing process by pinpointing parts of the software prone to defectiveness. This information can then be used to focus often limited testing resources, and to reduce time to find defects. Additionally, it helps in assuring software quality by locating defects that would not have been detected otherwise.

Software defect prediction consists of two areas, software metrics and classification. Software metrics are a wide collection of attempts at quantifying aspects of software and software development. The simplest metrics are for example lines of code in a file, or lines of code added per code update. Most of the software metrics have been developed with a specific goal in mind, such as measuring quality, cohesion or maintainability [2, 3, 4]. Interestingly, none of the software metrics in use for software defect prediction were specifically designed for defect prediction by machine learning techniques.

Classification in machine learning is the process of categorizing data into classes, for example sunny and raining. It is a type of supervised learning, meaning that when training the classification model, there are correct prediction answers available. In software defect prediction, the classification problem is often binary, that is each data point is classified as either defective or non-defective [5, 6]. Software defect prediction can also be non-binary if the goal is to predict the number of defects, however this case is not considered in this thesis. Classification became popular for software defect prediction studies after 2005, when several data sets containing software defect data were released [1]. Since then, several different classifiers and configurations have been tested to determine the best configuration. However, so far the results have been inconclusive.

This thesis will focus on a practical implementation of software defect prediction for RELEX Solutions. The aim is to implement and evaluate a software defect prediction framework based on the existing studies conducted in the field of software defect prediction. While several studies with similar goals have been conducted, the software defect prediction frameworks have previously often been assessed only by classifier performance metrics. As the evaluation has focused on classifier performance metric evaluation and comparison, the practical use cases have not been

considered, or have received less attention. In this thesis, focus will be on evaluating both the theoretical and practical performance.

The framework implementation process begins with identifying and presenting the use case for software defect prediction in Chapter 2. Additionally, the software release process at RELEX Solutions will be presented. Finally, the research questions are defined and described.

Next, the software metrics used for defect prediction are introduced in Chapter 3. Two types of software metrics, code metrics and change metrics, are used. Chapter 4 then introduces the classifiers, which are the machine learning algorithms used to make predictions based on the metric data. Additionally, several data management techniques and classifier performance measurements are introduced.

The implementation of the software defect prediction is introduced in Chapter 5, which consists of data collection and classification tools. Additionally, preliminary classifier performance testing is done.

In Chapter 6, classifier performance improvement techniques are implemented and analyzed, based on the techniques presented in Chapter 4. Additionally, an analysis of classifier performance is conducted, in addition to which the results are compared to other similar software defect prediction studies. Finally, a use case analysis is conducted, to provide an estimate on the suitability of the implementation to the use case presented in Chapter 2.

# 2 Background and goals

The software defect prediction framework which is presented in this thesis is created for RELEX Solutions. The implementation process starts by first defining a potential need for the defect prediction framework, and then validating whether and to what extent the framework can be used. This Chapter first introduces RELEX from the point of view of release management, and describes the processes used. Then, the goals for the software defect prediction implementation are presented.

## 2.1 RELEX Solutions

RELEX Solutions is a software company offering a SaaS product for demand forecasting, automated replenishment, space planning and assortment optimization to retailers and wholesalers. Software development work on RELEX software is split into several teams, one of which is the Release Management (RM) team. The RM team is mainly responsible for testing and managing the releases of new RELEX software versions. The software defect prediction implementation described in this thesis is targeted for the use of the RELEX RM team. This Subchapter introduces the RELEX software architecture briefly, and then presents the release and testing processes of the RM team.

### 2.1.1 RELEX software architecture

Figure 1 depicts a high level overview of the components of RELEX software architecture. The software consists of JavaScript-based web client, labeled as "User's browser" in Figure 1, and a backend. The backend consists of an in-memory database and business-logic calculations, labeled as kernel. both of which are programmed in Java. Additionally, included in backend is a JRuby based interface which serves the UI and data to the client.

The software is deployed as a Web Application Resource (WAR) file, along with customer specific configurations, which include a Ruby-based database schema and functionality configuration, and Java-based data adapters. However, for this thesis, only the main software is considered for software defect prediction purposes.

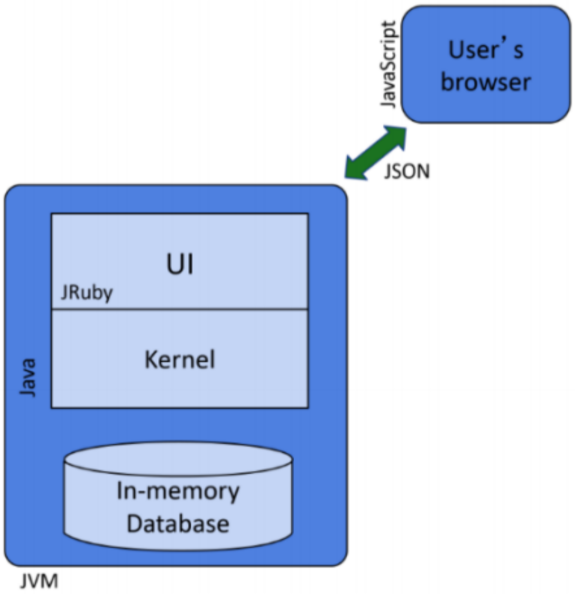


Figure 1: RELEX software architecture

### 2.1.2 Releasing and testing process in the Release Management team

The RM team releases a new version of the RELEX software at approximately three month intervals. The release process can be seen in Figure 2, where a version control system overview of the process is presented. Each dot represents a change in the software, and a labeled box represents the creation of a different branch of the software. The releases are numbered with version numbers, with the format x.x, and each new release incrementing the value by 0.1. For example the release following 6.5 would be 6.6.

The process starts with the creation of an alpha branch of the software, for example 6.5-alpha as in Figure 2. The alpha version is tested, and any defects found will be reported and later fixed by the development team, to both the Master and the 6.5-alpha branches of the software. Not all found defects will necessarily be fixed for the current version, some can be left to be fixed in later versions. The alpha phase lasts two to three weeks, after which a beta branch of the product is created, labeled 6.5-beta in Figure 2. Alternatively, if the alpha proves to be too defective, a new alpha branch can be created later, in which case the process would start over.

In the beta phase, the version release is further tested for defects. Additionally, the beta version will be subject to customer implementation specific testing. As in

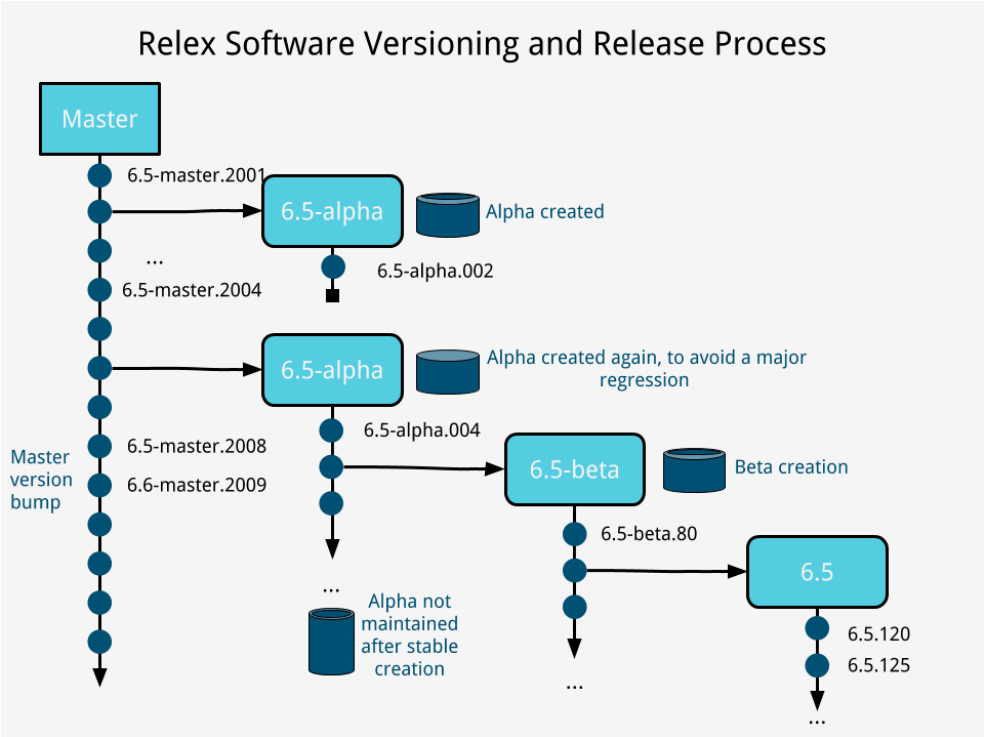


Figure 2: The life cycle of a RELEX software version

the alpha phase, any found defects are reported and some are fixed. This phase lasts approximately three weeks. The release branch is created when beta testing is over, and when release-blocking defects are fixed. The release-blocking defects are such known defects in the software that have been categorized by the RM team as blocking. The criteria for the categorization is that the defect prevents calculation in a core feature, or prevents normal usage of the software.

The new release version is released as a branch of the software, labeled 6.5 in Figure 2. When the new release version is released, a roll-out of the version to all customers is started. However, not all customers update immediately, or possibly at all to a new release. It is normal that a customer might skip a version and update later to a newer release. Some defects found at this stage are still fixed for the current version.

The testing of the product is split into three categories. Firstly automated testing, which features unit and unit integration testing, is used when making any changes to the software. This category also includes, for example, black box testing and end-to-end testing, which are not run as often as unit tests. Second, the RM team performs manual testing on the product, which is targeted by testing the parts thought to be most vulnerable to defects. This testing is mainly done at for alpha and beta branches of the software, and the beta phase customer testing is part of the process.

Other teams than RM also do some amount of manual testing, for example project delivery teams and business support teams. Finally, performance testing is done via specialized tools by the development or the RM team.

## 2.2 Current situation and goals

The current goal for the RM team is to improve the quality of version releases. Past versions, especially 6.2, have contained more defects on release than is desired. However, automated testing can only capture a certain amount of defects, and comprehensive manual testing would require additional limited human resources.

One solution would be to target human test resources more effectively. If testers would know more precisely which parts of the software to test, it would considerably limit the amount of resources needed for manual testing, and improve defect detection rate. This leads to the first Research Question (RQ) of this thesis:

> **RQ1:** Can software defect prediction be used to improve testing process in practice?

Furthermore, RM needs to decide the point in time at which an alpha version of the product is created. If the alpha is created at a time when the desired features for a release are present but have not yet been fully tested, testing in the alpha phase will be more challenging, and more defects can end up in the release of the version. On the other hand, it is important that new releases are released in time. This leads to the second research question of this thesis.

> **RQ2:** Can software defect prediction assist in choosing an optimal time for release?

The second research question is closely related to the first one. An optimal release time from defect prediction point of view is when the number of defects in the system is minimal. A well functioning software defect prediction implementation will provide an estimate on how many defects the system contains, which would affect the timing of creating an alpha version, or releasing a new version. This thesis will attempt to answer the research questions by implementing a software defect prediction for the defined purposes.

# 3 Metrics in software defect prediction

Software metrics are any measures that define quantitatively a property of a software. Software metrics have been designed and used for various purposes such as estimating quality or complexity [4, 7].

In software defect prediction, software metrics are generally used to predict defective components in a software, and in some cases also defect density. Most metrics however attempt to quantify other software qualities than defect proneness, such as cohesion, coupling or added lines of code [2, 8]. Thus, the usefulness of a software metric in this case is determined by the correlation between the metric and the defectiveness of the measured part of the software, rather than the values reported by the metric itself. Nevertheless, it is important to evaluate what the metrics with the highest correlation with defectiveness measure, to be able to better choose the data set and further develop metrics for software defect prediction.

This Chapter presents an overview on some of the commonly used metrics in software defect prediction studies. The selection of metrics is based mainly on which metrics have available data collection tools, which are detailed more in Chapter 5. Additionally, the selection is based on the success of the metric collections in software defect prediction studies [1]. The metrics presented here can be divided into two categories. The first is code metrics, which measures various attributes of the code. The second category is change metrics, which measures changes in the code of the software over time.

## 3.1 Code metrics

Most code metrics have been introduced as collections. When referring to the metrics, the names of the collections are normally used. The collections in turn are often named by the authors of the respective papers that introduced the metric collection. Several collections have been used for software defect prediction, however some collections have gained more popularity than others. These are presented below.

The most popular collection is the CK metrics collection [1, 2], which features several object-oriented metrics. The CK metrics extended [3] complements the CK metric collection by adding metrics to account for features the CK metrics do not measure. The QMOOD metrics [4] introduce a quality- and object-oriented comprehensive metric suite, featuring four different levels of metrics. Martin's metrics [9] is an

attempt to measure the stability and reusability of the code. Finally, McCabe's cyclomatic complexity metric [7] measures the complexity of the code from the different execution paths it can take.

The following Subchapters will present a more in-depth look into each of these metric collections, discussing the motivations behind each metric, and the pros and cons of their usage.

### 3.1.1 CK metrics

CK metrics, short for the names of the authors of the paper, Chidamber and Kemerer (C&K), is a collection of metrics introduced in 1994 [2]. In their paper, C&K scrutinize the existing software metrics based on the lack of theoretical basis, and the applicability of older non-object-oriented metrics to the object-oriented software analysis. In response, they designed a set of object-oriented metrics that aim to be theoretically solidly grounded.

**Weighted methods per class (WMC)** The WMC metric is defined as the sum of complexity values for each method in a class. As an example, if a class has $n$ methods and the complexity value for each method is 2, then $WMC = 2n$. This metric leaves the definition of method complexity intentionally open for interpretation.

C&K reasoned that this metric would provide an overview on how difficult developing and maintain the class in question is, due to the complexity of the class represented by the metric. Additionally, the metric shows the number of methods in the class, which impacts any children the class has due to the children inheriting all the methods. Finally, the larger the number of methods, the more likely the class is application specific, limiting reuse. Overall, a high WMC value is thus considered worse than low values.

WMC metric has been criticized for being ambiguous in definition [10], and having a dual purpose [11]. The two purposes are the complexity of the class as summed by the complexity of each metric, and counting the amount of methods. As the purposes do not correlate, the interpretations can cause difficulties in the usage of metric, depending on how the metric is used. To solve this issue, Li proposed that the metric should be split as two different metrics altogether.

**Depth Inheritance Tree (DIT)** The DIT metric is the length of the inheritance tree of a class starting from the highest level object. For example if class A inherits B, and B inherits C, then the DIT value for class A is 2. In many languages such as Java, all objects inherit at least the Object class, therefore making the minimum DIT value 1 for any given object.

DIT was created to represent the complexity of a class. The deeper a class is in the inheritance tree, the more methods it has likely inherited from its parent classes, making the class more complex. Additionally, the longer the inheritance trees are, the more complex the overall design is likely to be, but the more likely the methods of a parent class are to be reused. A high or low DIT value represents both good and bad qualities of the software, depending on which qualities are desired.

DIT has also been criticized of having unintended ambiguity in its definition [11]. The definition of the length of the tree is unclear if there can be multiple roots for the tree. Additionally, if multiple inheritance is in use, the length to the root, and the number of ancestors the class has is no longer the same.

**Number Of Children (NOC)** The NOC metric measures the immediate children of a class. To provide an example, class A that is inherited by classes B and C has a NOC value of 2, no matter how many classes inherit B and C in turn.

The basis for the NOC metric, as argued by C&K, is to measure class reusage, which correlates with the number of children of a class. On the other hand, if the class has a large number of children, it may indicate bad sub-classing and be a detrimental to the quality of the software. Furthermore, the more children a class has, the more influential the class is, which makes changing the class more difficult. An optimal NOC value should be balanced, rather than from either of the extremes.

While NOC has not received as much critique as the previous two metrics, Li questions why only immediate children are accounted for, instead of the whole inheritance tree [11]. Li argues that the class that is inherited from has influence over all descendant classes, and not only immediate inheritances.

**Coupling Between Object classes (CBO)** The CBO metric measures the number of classes a class is coupled to, where coupling is defined as one object acting upon other object. An example of coupling would be a method of class A using the method of class B, where both classes would have CBO value of 1.

CBO represents mainly design modularity, as the more couplings a class has, the less modular the design can be, and the less the class can be reused. Additionally, a class with more coupling is more prone to break when changes to other classes are made. Coupling also affects testing, making it harder to cover all cases the more inter-object coupling there are. Low CBO values are desirable, however, some coupling is considered good.

CBO has also been criticized for its ambiguity [11], with Li noting the lack of one standard for class coupling. Other coupling measures include inheritance and message passing.

**Response For a Class (RFC)** The RFC metric measures the response set of a class. C&K define response set as "the set of methods that can be potentially executed in response to a message received by an object of that class".

RFC captures the effect where if many methods are invokable from a class, the complexity of the class is likely to be higher. Additionally, it makes testing more difficult by requiring more understanding of the functionality by the tester.

RFC was cited by Li as being one of the more straightforward metrics [11], and no criticism or improvement suggestions to this metric was offered.

**Lack of Cohesion in Methods (LCOM)** The LCOM metric estimates the lack of cohesion in the methods of a class. Cohesion is defined by the absence of shared instance variables between the methods of a class. Lack of cohesion is calculated by subtracting the amount of methods that are cohesive, from the total amount of methods that are not cohesive. As an example, consider a case where method A uses variable set a,b,c,d,e, method B uses variable set a,b,e and method C uses variable set x,y,z. Each of the methods is compared with all other methods in the class, and cohesion is determined by whether the intersection of the instance variables is nonempty (cohesive) or not (non-cohesive). In this case, the LCOM metric is 1, due to A and B being cohesive, and C not being cohesive with either, resulting in $2 - 1 = 1$.

C&K note that cohesiveness in a class is desirable, due to encapsulation. Additionally, if a class is not cohesive, it should probably be split into new classes. Finally, low cohesion adds to complexity of the class.

The LCOM metric has been a subject of interest in many studies, and it has been

revised several times, including by C&K themselves, producing new versions of the LCOM metric [11, 12]. An example of the newer LCOM metrics is a metric called **LCOM3**. The new LCOM metric attempts to measure the same concept, but using graph theory to aid in defining cohesiveness. LCOM3 is calculated by forming a graph where the methods are the vertices, and an edge is formed between vertices if the methods share at least one variable. Then, LCOM3 = | connected components of graph |.

### 3.1.2 CK extended metrics

The CK extended metrics were introduced to complement the CK metric collection [3]. Tang et al. approached the CK metric set from a validation point of view. Their focus was to validate the CK metrics from fault predictiveness point of view. They found several aspects of software measurement that the original CK metric collection did not take into account.

Firstly, CK metrics do not take complexity sufficiently into account. Secondly, the dynamic behavior of the software is not considered, as the impact of classes that are used more frequently during execution is not taken into account in the CK metrics. Thirdly, in addition to direct inheritance, also indirect inheritances should be considered. This is the same notion that Li brought up in her criticism on the NOC metric [11]. The reasoning by both authors was that indirect children also have considerable impact and should be taken into account. Fourth, the relationship between inherited and new methods is not considered in the CK metrics. Tang et al. define that a method is dependent on another method, if the original method uses data which is modified or defined by the other method, thus making the original method dependent on it. The idea behind the concept is that if a new of redefined method modifies data that is used by a redefined method, it will affect the defect-proneness of the inherited method. Finally, the classes with more memory or object allocations cause more faults in the software, which is not represented by CK metrics.

Based on the criticism presented on CK metrics, four new metrics were presented to add to the existing CK metrics collection.

**Inheritance Coupling (IC)** The IC metric targets the fourth critique on CK metrics. IC counts the number of parent classes the target class is coupled to. In this metric coupling is defined as such that a class is coupled to its parent if any of the methods of the parent class are functionally dependent on new or redefined

methods of the target class. Functional dependency is defined as such that a new or redefined methods affects data used by an inherited method.

**Lack of Cohesion in Methods (CBM)** CBM further defines the relationship of inherited and new or redefined methods between a class and its parent classes. CBM counts the total number of such methods in a class that are coupled to the methods of the parent classes. The metric is very closely related to the previously presented IC metric, with the difference that this metric takes into account method level count of the couplings, while IC more abstractly only counts it on class level. Furthermore, CBM takes better into account the increased complexity of having more methods coupled.

**Number of Object or Memory Allocations (NOMA)** The NOMA metric addresses directly the concern for measuring memory allocation. It counts the total of all statements that allocate memory in a class. However, indirect allocations are not considered, such as calling another method.

**Average Method Complexity (AMC)** The AMC metric is the average of the size of the methods of a class. The authors leave the exact definition for size open for interpretation, but a simple measure such as lines of code could be used here.

### 3.1.3 QMOOD metrics

The Quality Model for Object Oriented Design (QMOOD) metric collection is a quality-oriented attempt at creating a comprehensive standard for describing object-oriented software [4]. The model consists of four levels. The highest, first level defines overall quality attributes of a software, for example reusability and flexibility. The second level defines design properties, which are for example hierarchies and coupling. Third level of QMOOD defines design metrics, which are the concrete software metrics. Finally, the lowest level is the fourth level, which defines design components of the target architecture. These in practice refer to the code itself.

All of the levels in QMOOD are directly related to the level above it. To provide an example, the fourth level is used to collect data for the metrics of the third level. Then, each metric is mapped to a design property of level 2, so that for instance Coupling is the Direct Class Coupling (DCC) metric. Finally, based on the values

of level 2, the quality attributes can be calculated by the formulas provided in the paper. For example, Reusability is defined as $Reusability = -0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$.

For defect prediction, only the level 3 of QMOOD model is used, due to the higher levels values being derivations of the metrics defined in the third level. Despite this, the full model helps to understand what was intended purpose of the level 3 metrics. In total, QMOOD level 3 metric consists of eleven metrics.

Due to metric collection tool limitations, the software defect prediction implementation in this thesis uses only some of the metrics defined in the QMOOD metric set. The metrics not selected will not be covered here.

**Data Access metric (DAM)** The DAM metric describes QMOOD level 2 Encapsulation property. Encapsulation in object-oriented programming refers to qualities such as class variable and method hiding, which are in Java for example protected or private variables and classes.

Based on this description, the DAM metric in practice is defined as being the ratio of private and non-private variables within a class. Higher DAM values are more desirable, meaning the more encapsulation there is the better the quality is. DAM values range between 0 and 1.

**Measure of Aggregation (MOA)** The MOA metric measures Composition of the QMOOD level 2 attributes. Composition is defined as the measure of so called part-whole relationships, which is the amount of an entity participates in the whole, and which entities the whole consists of.

To measure the part-whole relationship in practice, MOA uses the attributes of the measured class. It counts the sum of attribute declarations, where the type of the attribute is a class defined by the user.

**Measure of Functional Abstraction (MFA)** For the MFA metric, the corresponding level 2 design property is Inheritance. In QMOOD Inheritance is defined as "is-a" relationship between two classes, and relates to the level of nesting of classes in the inheritance hierarchy.

This relationship is quantified in the MFA metric as a ratio of the inherited methods of the target class, to the amount of methods accessible from a method in the target

class. The value range for this metric is from 0 to 1.

**Cohesion Among Methods of Class (CAM)** The CAM metric measures the Cohesion of the QMOOD level 2 design attributes. Cohesion is defined in QMOOD similarly to the cohesion defined by C&K, in which cohesion is the measure of relatedness between the methods of a class.

To calculate the CAM metric, first the the sum of the number of different types of method parameters in each method is taken. Then, the acquired sum is divided by the multiplication of total number of different method parameter types and total number of methods in the target class. The resulting value represents the relatedness among the methods of the class. Values range from 0 to 1, where values closer to 1 are preferred.

**Class Interface Size (CIS)** The corresponding QMOOD level 2 design property for CIS is Messaging. In QMOOD definitions, Messaging is the measure of the services that the class provides. For the CIS metric, this is simply the count of public methods in the measured class.

### 3.1.4 Martin's metrics

Martin investigates in his paper what makes code stable and reusable [9]. His main focus was on interfaces. He considered an example where a keyboard reader and printer writer are used by a copy class. Then, the reader and writer are split into reader and keyboard reader, and writer and printer writer respectively. Martin argues that this provides better generality and reusability. Martin notes that the new reader and writer classes are highly unlikely to change. Furthermore, the stability of the interfaces makes for a good dependency.

Based on these observations Martin attempted to create a metric set that would measure the independence, stability and responsibility of a class. If a class does not depend on any other class, it is independent. If a class is relied on by other classes, it is responsible. Stable classes are both responsible and independent. These three qualities measure the role of a class from interfacing point of view.

In total Martin created five metrics, of which only two are commonly used in software defect prediction studies. Since the three other metrics are rarely used, they are covered here only briefly, and will not be used for defect prediction in this study.

**Afferent and Efferent Couplings (Ca and Ce)** The two main metrics that were introduced are Ca and Ce. Ca measures the number of classes that depend upon a class, while Ce measures the number of classes a class depends upon. The two metrics are directly related to Martin's theory of responsibility and independence, and quantify the interface relationship of a class.

On the usage of these metrics, Martin warns that the usage of the metrics as strict guidelines is not advised, and the appropriateness of the metrics will most likely vary case to case.

**Other Martin's metrics** The third of Martin's metrics, instability (I), measured the combination of Ca and Ce, representing the stability quality. Fourth metric is Abstractness (A), which measures the rate of abstract classes in to the total classes, and the final metric combines I and A to create the Distance (D) metric.

### 3.1.5 Other metrics

This Subchapter covers the few metrics that are not part of any collection, but are nonetheless used widely in software defect prediction studies.

**Cyclomatic Complexity (CC)** McCabe's CC metric is one of the oldest software metric used for defect prediction, introduced already in 1976 [7]. The idea behind the metric is straightforward, it measures the linearly independent paths that a program can take. The more modern use case for program is CC in a method, which can be averaged to produce the average CC for a class. More precisely, CC is calculated by forming a graph from the paths of the program or function. The formula provided by McCabe for CC is $v(G) = e - n + 2 * p$, where $e$ is the number of edges, $n$ is the number of vertices, and $p$ is the number of connected components. To provide an example, the CC value for a single if-else function would be $v(G) = 4 - 4 + 2 = 2$. The value of CC is always at least 1.

The CC metric has been used for many different purposes, such as unit-testing effort [13], but the results have been mediocre at best. CC has also been criticized for being too strongly correlated with LOC, thus making it simply a convoluted way of measuring size of a method.

**Lines of Code (LOC)** The LOC metric is arguably the simplest code metric. It is not specifically introduced as a part of any collection or study, but it is often used in addition to other metric collections [1]. It measures the lines of code in a defined target, which is often a method, class or file. Several variations of LOC exist, for example Java code lines could be counted from either Java bytecode ".class" files, or Java code lines from ".java" files. Furthermore, if code files are used, variations include whether to include lines with only line breaks, or comment lines.

Despite the relative simplicity of LOC, it has had good success in defect-prediction studies. This is explained by the fact that the largest modules tend to have the most faults, with one study citing that 20% of the largest modules containing 51-63% of all defects [14].

## 3.2 Change metrics

Change metrics measure changes in the code of a software over time. The existing change metrics in defect prediction literature are not as well-defined as code metrics, and they are used in fewer software defect prediction studies. Despite this, studies comparing change and code metrics have achieved results where change metrics outperform code metrics [8].

Arguably the greatest benefit of change metrics over code metrics is the language agnosticism of change metrics. Furthermore, version control systems are widely in use, making change data readily available. This makes change metrics in many cases more accessible than code metrics.

While there are no generally used collections for change metrics, some basic metrics are often same between different studies. In this thesis, the metrics defined by Moser et al. are used [8]. Additionally, the extension to Moser's change metrics defined by Choudary et al. will be covered [15]. While the latter are not used for defect prediction in this thesis, the paper provides good insight into the change metrics overall.

### 3.2.1 Moser's change metrics

Moser et al. hypothesized that change metrics contain more information on the defectiveness of a file than code metrics [8]. To test the hypothesis, a collection of change metrics was created . These were then tested against selected code metrics,

where promising results were achieved with the new change metric collection.

Two of Moser's change metrics are not used in this defect prediction implementation, as those rely on heuristics to extract values. These are Bugfixes and Refactorings, which are extracted by analyzing commit messages from version control systems. The rest of the change metrics are presented below.

- **Revisions:** The number of separate changes made to a single file.
- **Authors:** The number of unique authors that made changes to a file.
- **LOC added:** The sum of lines of code added to a file for all revisions. It is also used to create metrics Max. LOC added and Avg. LOC added, instead of using sum.
- **LOC deleted:** The sum of lines of code deleted from a file for all revisions. It is also used to create metrics Max. LOC deleted and Avg. LOC deleted, instead of using sum.
- **Codechurn:** The sum of LOC deleted, and LOC added, where the results of this calculation are then summed over all revisions of a file. It is also used to create metrics Max. Codechurn and Avg. Codechurn, where sum is replaced with avg. and max. respectively.
- **Changeset:** It is the number of files per change, used as Avg. Changeset and Max. Changeset. To use it on file level, each file in a commit is added the value of change set for Avg. Changeset, which is at the end of data extraction divided by the number of revisions for the file. To calculate Max. Changeset, the file receives the largest commit change set value which the file was a part of.
- **Weighted age:** The value of Weighted age metric is the number of weeks between the first and the last changes made to a file.

### 3.2.2 Choudary's extension to Moser's change metrics

Choudary et al. continued developing change metrics based on Moser's change metrics [15]. While the Choudary's extended set is not used for defect prediction in this thesis, their work provides some valuable insight into change metrics overall.

In addition to the new metrics, the perhaps more interesting contribution in the paper is a categorization for change metric types. Four categories are introduced. The first category is standard change metrics, which includes for example LOC added, LOC deleted and other similar metrics that measure direct changes to the code. These are expected to have direct relationship with defect proneness. The next category is developer-based change metrics, which includes metrics such as LOC added per developer and codechurn per developer, both of which are new metrics introduced in Choudary's metrics. Additionally, Moser's Authors metric would fit into this category. The developer based metrics as the name suggests are extracted per developer, not per code change as the other change metrics. The third category is period-based change metrics, which includes metrics such as Weighted age, or the new Choudary's metric time-difference between commits. These metrics measure intervals between changes, as opposed to types of changes. It is expected that smaller change intervals cause more defectiveness in a file. Finally, the fourth category is uniqueness-based change metrics, which contains metrics which attempt to measure whether the change was unique to a file. This category contains only the newer Choudary's metrics, such as the single commits metric, which measures the number of commits where a file was committed alone.

# 4 Classification in software defect prediction

Classification in machine learning is the process of separating data items into categories, based on a training data set given to a classifier. Data for classifiers is separated into independent variables, which are the explanatory features for classification, and a dependent variable, which is the value the classifier attempts to predict. The dependent variable is often also called the class variable. In software defect prediction, the class variable is binary, and the two values are defective or non-defective. Each prediction is made as a confidence percentage, which represents the probability of the data item being positive.

This Chapter introduces how classification has been used in software defect prediction studies. In addition to the classification algorithms, the classification process, including data preprocessing classifier performance measurement, is likewise covered.

## 4.1 Measuring classifier performance

Measuring classifier performance is done using four measures of classification correctness. These four measures are the number of True Positives (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN). The four measures constitute the confusion matrix as seen in Table 1, from which other measures of classifier performance are derived from.

Table 1: Confusion matrix

| **Confusion matrix** | Condition true | Condition false |
|---|---|---|
| Prediction true | TP | FN |
| Prediction false | FP | TN |

Each of the four measures in the confusion matrix defines an aspect of the correctness of the results of a classification. The formal definitions for each of the measures are the following:

**TP:** True positive is the number of the result rows of a classification, where the actual value of the class variable is positive, and the predicted value is also positive. In software defect prediction, a positive value refers to defective files. The consensus in classification studies is that the minority class is set as the positive value in the confusion matrix.

**FP:** False positive is the number of classified result rows where the actual value the class variable is negative, but the predicted value is positive.

**TN:** True negative is the number of classified result rows where the actual value of the class variable is negative, and the predicted value is also negative. Here negative refers to files that are non-defective or *clean*.

**FN:** False negative is the number of classified result rows where the actual value of the class variable is negative, but the predicted value is positive.

An important concept regarding the measures in the confusion matrix is the cutoff-point value. The cutoff-point defines the probability threshold above which classification result rows are considered positive and below which result rows are considered negative. The values of cost matrix will differ based on the chosen cutoff-point value. For example, if cutoff-point is 0.0, then all results are either true positive or false negative. The default cutoff-point value is 0.5 unless specifically otherwise stated. In addition to affecting the confusion matrix measures, cutoff-point value also affects the measures derived from it.

There are several different measures that can be derived from the confusion matrix. Each of these measures a different aspect of the results of a classification. The most prevalent and relevant measures to this thesis will be reviewed in the rest of this Chapter. Due to each measure having several accepted names, the most common names in general use will be presented here, and the name in the title will be the one used in this thesis. A summary of the formulas for each of the measures can be found in Table 2.

The first measure is *Accuracy*. It is also known as correct classification rate, and is arguably the most intuitive measure of classifier performance. It measures the percentage of results that have been correctly classified, including all four measures of confusion matrix in the calculation. While accuracy gives an overview of classifier performance, its values can, however, be often very misleading. For example, consider a classifier that classifies all class variables as negative. If 950 out of a 1000 input rows are negative, then the accuracy for the classifier is 95%, even though the classifier essentially did not predict anything.

*True Positive Rate (TPR)*, also known as recall or sensitivity, is the second measure that can be derived from the confusion matrix. The value of this measure is the probability that a positive data row will be predicted as positive. In other words,

Table 2: Confusion matrix

| Measure | Formula |
|---|---|
| Accuracy | $\frac{TP+TN}{TP+TN+FP+FN}$ |
| TPR | $\frac{TP}{TP+FN}$ |
| TNR | $\frac{TN}{TN+FP}$ |
| FPR | $\frac{FP}{FP+TN}$ |
| FNR | $\frac{FN}{FN+TP}$ |
| PPV | $\frac{TP}{TP+FP}$ |
| F1-measure | $\frac{2TP}{2TP+FP+FN}$ |

it is the ratio of correctly predicted positive results to all results which have actual positive values. TPR is not as generally applicable as accuracy as a classifier performance measure. Instead, it measures a specific quality of the performance of a classifier, which it does well. Despite this, also used widely as a performance measure for classifier comparisons [1].

*True Negative Rate (TNR)*, which is also called specificity, is similar to the TPR measure. TNR measures the probability of a negative data row being classified as negative, while in comparison TPR predicts the same for positive rows. Furthermore, like TPR, TNR is also a specific measure rather than an overall classifier ranking measure.

*False Positive Rate (FPR)* or fall-out, is the fraction of actual positive data rows that are predicted as negative. Similar to TPR and TNR, FPR is not very well suitable for overall classifier performance analysis.

*False Negative Rate (FNR)* or miss rate, is similar to FPR, but with the classes other way around. It measures the fraction of actual negative data rows that are predicted as positive.

*Positive Predictive Value (PPV)*, also known as precision or correctness, is the proportion of actual positive data rows in all data rows that were predicted as positive. This measure can be seen as an accuracy measure for positive rows only, which makes it valuable as a performance measure if only positive classification results are

considered, as is often the case in for example software defect prediction. However, in general, it is not a good overall benchmark for classifier performance, as it suffers in part of the same problems as accuracy.

*F1-measure* or F-measure is an attempt at providing an overall measure of a classifier's performance. It is calculated as the harmonic mean of PPV and TPR.

*Area Under the Curve (AUC)* is another attempt at an overall measure for classification performance measurement. AUC is derived differently from the confusion matrix as the other measures. It is calculated by first plotting the values of FPR and TPR at each cutoff-point value on x and y axises respectively. The resulting curve is called *Receiver Operating Characteristics curve (ROC)*. The AUC value is then calculated as simply the area under the ROC curve. The AUC value ranges from 0 to 1, where 0.5 is the baseline, indicating the classifier is outputting arbitrary results, and 1 indicating a perfect classification. AUC has been proposed as the primary measure for classifier performance measurement in defect prediction over the other presented measures [5]. Regardless it is not used as often as some of the other presented performance measures [1]. One of the key benefits of AUC is that it is not dependent on choosing a cut-off point, as the other measures are. This increases the stability of the AUC measure as comparison tool, especially between different studies.

## 4.2 Overview of classifiers

Software defect prediction studies have experimented with a wide range of classifiers in an attempt to find the best performing classifier for defect prediction [1]. However, findings on which classifier performs best varies from study to study. Because of this, no conclusive results on which classifier performs best has been achieved. Instead, results implicate that the classifier should be chosen per case basis, depending on which measures of classifier performance are emphasized.

In the rest of this Subchapter the classifiers that have overall been seen to have good performance will be reviewed.

### 4.2.1 Random Forest

The Random Forest (RF) algorithm is one of the best-performing and most often used classifiers in software defect prediction [1, 5, 16, 17, 18]. It is a tree-based en-

semble classifier introduced by Leo Breiman in 2001 [19]. The RF classifier functions by combining votes from a collection of decision-trees to make its classification.

The popularity of the RF classifier is due to several factors. Firstly, it is is easy to use, in part due to its resilience to outliers and noise in the data, and in part due to its ease of configuration [1, 17]. Additionally, RF includes functionality to identify important parameters from the data, which increases prediction performance. Finally, classification with the RF classifier is fast, which makes it optimal for large data sets. In conclusion, while RF might not always the best performer, it scores consistent results in terms of AUC values and ranks usually at least close to the best performing classifiers.

### 4.2.2 Naive Bayes

The Naive Bayes (NB) classifier is a simple, statistical-based approach to classifying. It is a well-known classifier that is used in other areas as well, such as text classification and medical diagnosis [20]. The predictions for the NB classifier are calculated for each of the attributes independently by applying the Bayes rule for calculating probability of the class based on the attribute instances [21]. The simplicity of the NB classifier comes from an assumption that the features provided to the classifier are independent from each other. This makes it efficient, but naturally it does not take into account feature correlation.

In defect prediction, NB is considered a benchmark for whether a more sophisticated model is useful for classification or not, as NB is relatively simple compared to other classifiers. Despite its simpleness, NB has also consistently achieved acceptable performance in classification studies [1, 5], sometimes achieving best performance compared to other, for example RF, classifiers in terms of AUC [16].

### 4.2.3 J48

The J48 algorithm is an open source implementation of the C4.5 decision tree classifier. The J48 classification algorithm forms decision trees with certain guiding principles, and the results are presented based on the constructed tree [22]. J48 decision trees can also be pruned to generalize the tree, after the main algorithm has created the tree. Pruning reduces outliers, thus reducing classification errors.

J48 has achieved good results in defect prediction studies [1], in some surpassing the performance of RF for example [6]. Despite this, the results have arguably not

been as consistently good as other algorithms, such as RF or NB.

### 4.2.4 Support Vector Machine

Support Vector Machine (SVM) is a sophisticated maximum margin classifier introduced in 1995 [23, 24]. SVM classifier functions by attempting to separate the data points by a division where the difference is of maximum width. SVM behavior can be modified with a kernel function that maps each dot product into a higher-dimensional feature space, which has the benefit of the data being more easily linearly separable.

SVM has had varying success in defect prediction studies. A few studies advocate strongly for SVM use, presenting good results achieved with SVMs [23, 24]. However, in total SVM has had less success than most other popular classification algorithms [1].

### 4.2.5 Bayesian Network

Bayesian Network (BN) classifier is an evolution of the NB classifier [21]. It is an attempt to avoid assuming variable independence in the classifier, which is a main criticism of the NB classifier. The technique leverages Bayesian networks to encode independence statements for the variables.

In defect-prediction BN is quite rarely used [1]. Despite this, it has had acceptable results and can perform better than some of the more sophisticated classifiers, such as J48 or RF.

## 4.3 Enhancing classifier performance

Besides choosing a best fitting classifier, there are several ways to improve classification performance. This can be done either by manipulating the input data of a classifier, or by using a meta-classifier with the originally selected classifier to enhance the results.

Several of the presented techniques below require a certain amount of manual trial and error to achieve the most suitable values. This presents the danger of overfitting the model to only one use case or even to a single data set. Overfitting in classification happens when a classifier is tuned too much for a specific training data set,

decreasing performance when the classifier is applied to broader data sets. Overfitting should be avoided when using these techniques by using as generic setting as possible while maintaining good results over multiple training data sets.

### 4.3.1 Data preprocessing

Data preprocessing is arguably the simplest way of improving prediction accuracy. This category of performance improvements refers to data quality improvement and applying different data filters. Data quality can be improved in several ways, including removing outliers and dealing with missing values in the independent variables [25].

Filtering refers to a function that can be applied to the data to transform it. As an example *log filtering* has been found to work well with some classifiers [5, 25]. Log filtering is a technique where all numeric values $n$ in the data are replaced with $ln(n)$ values.

Data normalization is another common data preprocessing technique. To normalize the data, each numerical value is converted to a value between zero and one. This reduces the impact of very large values to classifier performance.

### 4.3.2 Feature selection

Feature selection is the process of reducing the the independent variables to only a subset of the original. It has the benefit of reducing processing time and in certain cases enhancing classifier performance.

A type of feature selection which is popular in defect prediction is Correlation Feature Selection (CFS) [1], which was introduced in 2000 by Mark Hall [26]. This technique analyses which independent variables are least correlated with the class variable, and most correlated with each other. It then removes those independent variables from the data set. The idea is that the remaining data set contains less noise and gives better predictive accuracy.

Feature selection works best with less sophisticated classifiers that do not implement some form of feature selection on their own. For example, the NB classifier is a classifier where Feature selection has been found to perform well [5, 25], further improving the results achieved by the classifier.

Table 3: Cost matrix

| **Cost matrix** | Condition true | Condition false |
|---|---|---|
| Prediction true | 0 (TP) | 1 (FP) |
| Prediction false | 10 (FN) | 0 (TN) |

### 4.3.3 Over and undersampling

Class imbalance problem is a classification issue where one class is featured considerably more frequently in the data set than the other. This can cause the classifier to classify more heavily towards the more frequent class than what is desired.

This problem can be alleviated by Over- or Undersampling the data set [27]. In Oversampling, new rows for the minority class are generated until the classes are in balance. Undersampling accordingly removes instances of the majority class until the classes are in balance. Alternatively, Over- or Undersampling can balance the classes to a certain ratio, instead of one to one relation. The benefits of these techniques are their simpleness and effectiveness, however, the effectiveness can be dependent on the chosen classifier and data set. Additionally, the rate of Over- or Undersampling must be carefully chosen per case basis.

Overall, Undersampling is considered as the better of the two, and it has been proven not to degrade the results of classification even though it reduces the amount of data [28].

### 4.3.4 Cost-sensitive classification

Cost-sensitive classification is an alternative option to manage the class imbalance problem [8, 29]. Cost-sensitive classification functions by assigning a cost value for each measure in the confusion matrix. The result is a cost matrix, which contains the cost weight of misclassification for each measure type. An example cost matrix can be seen in Table 3.

The convention in cost matrix usage is for the values of TP and FP to be set to 0, since these represent correct classification. Additionally, if the class that is in the minority is the focus of the prediction, then the cost of FN should be higher that the cost of FP. Thus the cost for misclassifying the majority class can be set to one, and

the cost for misclassifying the minority cast is set to $n > 1$. Table 3 is an example of such a configuration. In practice, this setup aims to reduce the misclassification of the positive class.

### 4.3.5 Cut-off value

Choosing the cut-off value is a means to adjust the performance of a classifier. By default, the cut-off value is 0.5, with which predictions with a confidence value over or equal to 0.5 are seen as positive, and under 0.5 negative. Most studies use the default value of 0.5 for cut-off value [6]. While this makes it easier to compare the results of the studies, the default value is likely not the best option for each use case.

The chosen cut-off value can affect which metric set or which classifier is the best for a given use case. For example, consider a classifier that has a PPV value of 0.2 at cut-off point 0.5. The same classifier could have a PPV value of 0.6 at cut-off point 0.75. If there are many items of data, then the performance of the 0.75 cut-off value can be better for predicting only positive value, if a high PPV is desired, even if it captures fewer actual positive values. This applies notably if the data set is imbalanced [6]. Choosing a suitable cut-off value is difficult and must be chosen per case basis, by experimenting with different cut-off values.

### 4.3.6 Bagging and boosting

Bagging and Boosting are meta-classifiers which are used for enhancing the performance of a given classifier [30]. Both meta-classifiers work by manipulating training data to generate improved classifiers given the base classifier. The Bagging technique generates multiple training sets from the original by sampling with replacement, then the results are combined by voting. Boosting on the other hand uses training data as-is, but assigns different weights to instances. This training is repeated several times, each time adjusting the weights, causing the classifier to focus on different instances of the data. Finally, results from different iterations are combined by voting. An often-used implementation of boosting is the Adaboost.M1 classifier.

In software defect prediction, Bagging and Boosting have been used to enhance performance of some of the presented popular classifiers. For example, Adaboost with J48 was found to be the best performing classifier of the studied classifiers in a study by Wang et al. [27].

# 5 Implementation research

In this Chapter, the framework implemented for this thesis for data gathering, management and software defect prediction is introduced. The first step in this process is defining data sets, after which data can be collected. This is done via pre-existing tools, and collected from Java binaries and version control system data. Then, the data management and classification tools are implemented as command line tools in Java.

Additionally, a preliminary experiment and performance analysis on selected classifiers is conducted in order to narrow down the selection for analysis in Chapter 6. The experiment is performed on a set of five classifiers which were introduced in Chapter 4, with minimal configurations applied.

## 5.1 Data collection

Data collection is a vital part of software defect prediction. The performance of the classifiers can be severely improved or limited by the quality of the data set. The data collected for this implementation of software defect prediction can be split into two categories. First is the defect data, and the second is the software metrics data. All data is collected with the goal of combining it together to produce a final single file per each software version. This final version is then used to for defect prediction. Different data can be combined to form different final data sets. The RELEX software versions for which data is collected are 6.0 to 6.3. However, before collecting the data, some key decisions on data collection should be considered.

### 5.1.1 Defining required data

First consideration of data collection is the level on which the data should be collected. For software defect prediction, there are several options. Data can be collected for example on per class [6], per file [8], or per module [23] level. The implementation of software defect prediction in this thesis uses file level data collection. Thus, all data which are not on the file level must be aggregated to the file level.

Next, the desired data sets need to be defined. In this implementation, there are two cases that need to be considered when deciding data sets. Firstly, the data set for when an alpha branch is created, and secondly, a data set for when a release branch is created. A part of the data can overlap with both data sets.

To satisfy the requirements made in the definition, six files of data are collected. The first four are defect data for alpha and release, and code metric data for alpha and release. Change metric data for alpha is collected using historical change data from three previous versions, and using change data from alpha to release. These six files form one complete data set for a single version.

### 5.1.2 Extracting defect data

With the desired data sets being defined, the data collection can commence. The most important of the data sets is the defect data, without which the classifiers cannot be trained. Thus defect data collection is done first.

The primary concern when collecting defect data is defining what is considered a defect. This definition varies from study to study, with the most lenient definitions being simply collecting all commits from a version control system (VCS) that contain the word "bug". In this implementation, the definition for a defective file is any file that has had any changes made to it in a issue that is marked as "Dev - Bug" in the issue tracker used by RELEX. This process is similar to what was used by Gyimóthy et al. [31].

The process for collecting the desired defect data starts with extracting data from the issue tracker. RELEX uses Redmine as the issue tracker, in which an issue consists of an issue number and additional descriptions such as who the issue is assigned to, version number and other information. Any change made to the software should have a issue assigned to it. From Redmine, all issues from the whole period of development are extracted, filtering by issue type "Dev - Bug". The type field in Redmine portrays what type of development the issue required. Some examples of issue types are bugfixes, refactorings and feature additions. The data export is done manually from the Redmine web UI, but could in future be done automatically using Redmine API.

To link the issue data to bug fixes, it must be combined with VCS data. RELEX uses Git as the VCS. All changes to the software are generally made to separate branches, and when ready, the changes are squashed to a single commit. Squashing is a process in Git where several commits in a branch are combined as one. The squashed commits are the changes that are considered in this data collection. Each squashed commit should contain a issue number prefixed to it, which is the number of the issue that the commit is related to. The format is the following: "#12345:

Fixed bugs", where the number between the hash-tag and colon is the issue number.

Next step is then to combine the data sets from Git and Redmine. This is done using a Python script, which does roughly the following. It requires as a parameter the version for which version the defect data is collected. Then, it collects all commits from the alpha version branch that have been made until the release version. The same process is repeated for commits in the release branch, starting from the creation of the release branch, and ending in the latest commit to that branch. Then, the issue numbers in the commit messages are cross-referenced to the issue numbers in the list of bug-fix issues extracted from Redmine, and reduced to only those commits that have been made in response to a bug-fixing issue in Redmine. Additionally, a list of files that were changed in the commit can be extracted from Git.

Now there are two files with a list of files that contain defects. One for the when alpha of the version was created, and one for the when release was created. One more step is required to complete the data sets. For release defect prediction, the release defect list can be used as-is, but for alpha both release and alpha defects are desired. Thus, for the final alpha defect list, both defect data sets are combined together.

Overall, the defect data for each version contained at most approximately 550 rows of defective files, and at least 200 rows. The process for collecting defect data can also be seen in Figure 3. The figure contains the whole process of data collection, with the final defect files being the files prefixed with "files_with_defects". The defect data collection in this implementation is similar to the defect data collection process of other software defect prediction studies, for example the data collection done by Choudary et al. [15].

### 5.1.3 Extracting software metric data

The next step in data collection is the extraction of independent variables, which here refers to the code and change metrics. For this purpose, two existing metric collection tools are used to extract the two different types of software metrics. Both tools were chosen as they were the only readily available and suitable tools, which contained a wide array of the desired metrics.

For code metric extraction, the CKJM extended tool is used [32]. CKJM extended is a tool for extracting several code metrics from compiled Java bytecode ".class" files. It has also been used in other defect prediction studies, such as the study by

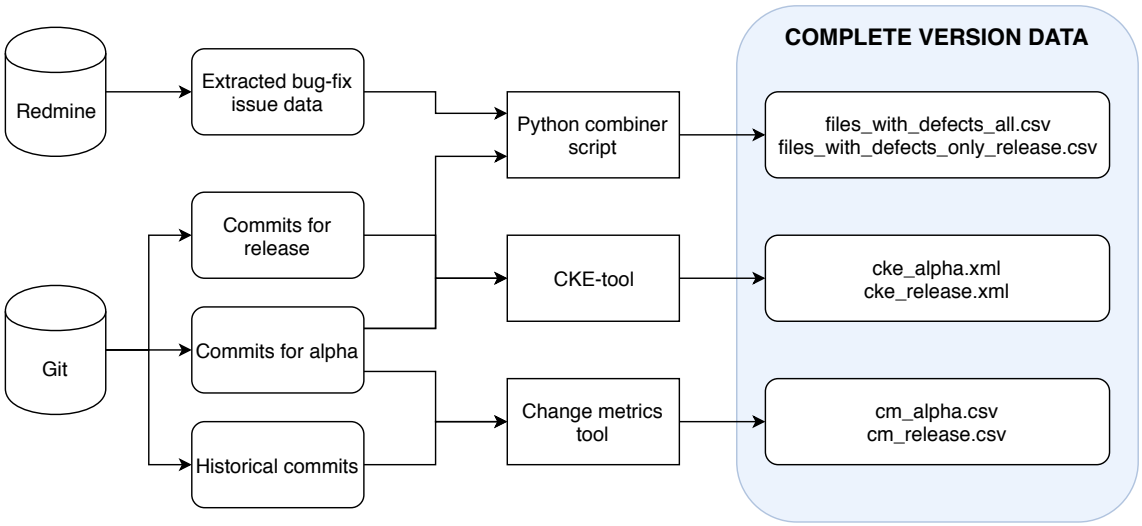


Figure 3: Data extraction for a single version

Malhotra et al. [18]. Due to using compiled Java bytecode as the input for metrics, CKJM extended can only extract data from software languages that are compiled to Java bytecode.

CKJM extracts in total 19 code metrics, which are the code metrics introduced in Chapter 3, disregarding the metrics that are specifically stated not to be included in the defect-prediction of this thesis. It should be noted that due to some of the definitions of code metrics being ambiguous, intentionally or accidentally, practical implementation decisions when creating metrics have to be taken. This is also the case with CKJM extended. The exact descriptions for how the metrics are implemented in CKJM extended can be found in the documentation provided on the CKJM extended project page [32].

The CKJM extended tool is used to collect two data sets for defect prediction for the two branches, alpha and release. A data set for alpha is created by setting the code base from Git to the point of alpha branch creation of the target version, compiling all Java classes in the project, and running CKJM extended. The same process is done for the point of release branch creation for the target version. In total, approximately 3000 files were included in both data sets.

For change metric collection, the tool created by Maurício Aniche is used [33]. The tool collects the change metrics defined by Moser from Git history [8]. While this tool can collect data from any file in the project, the files with no changes at all from the time period are not included in the data set. This is to prevent too many rows containing only zero values from degrading classifier performance.

In practice, the process for the extraction of change metrics is conducted by first deciding the amount of change history that is included. It is decided that three previous versions of change history are included in the data, to obtain change metric data for as many files as possible. Then, the change metric tool is used on Git commit range between the creation of the version three versions behind the target version, and the creation of the alpha version for the target version number. Additionally, the tool is run for the range of commits between creation of alpha version and creation of release version for the target version number. Generally, the change metric dataset includes over 5000 files.

The extraction process of code and change metrics is visualized in Figure 3, where the collected code metrics are the files with the prefix "cke_", and the collected files for change metrics are files with the prefix "cm_".

### 5.1.4 Defining final data sets

Out of the total six collected files, eight final data sets are created for defect prediction. Each data set is created out of one of the two defect data files, and a combination of the metric files. A final data set contains both independent and the dependent variables. Thus, a single final data set can be used as both training or testing data. To provide an example, if defect prediction is done for the alpha branch of the version 6.2 using CM, then the CM file for 6.1 is used as training data, and the CM file for 6.2 is used as testing data. The final data sets are intended for classifier performance analysis. In practical prediction tasks, the same dataset would be used, but without defect data.

Three of the data sets are for defect prediction at the beginning of alpha phase. These data sets use the list of defective files for the alpha phase only. The created data sets are listed below, with the files that are used to create the data set.

- **CM:** files_with_defects_only_release.csv, cm_alpha.csv
- **CKE (alpha):** files_with_defects_only_release.csv, cke_alpha.xml
- **CM + CKE:** files_with_defects_only_release.csv, cke_alpha.xml, cm_alpha.csv

The remaining five data sets are for defect prediction at software version release. For these data sets, the used defect data set is the complete list of defective files for a version. Additionally, as both change metrics files are used, change metric files

need to be unified into a single file when creating the final data set. This is done by either aggregating, in which case the weighted average is used for unifying each row where the file names match, or by combining, in which case the values are retained for both files in separate columns. The five created data sets are listed below.

- **CKE (release):** files_with_defects_all.csv, cke_release.xml
- **CM aggregated:** files_with_defects_all.csv, cm_release.csv
- **CM combined:** files_with_defects_all.csv, cm_release.csv
- **CM + CKE aggregated:** files_with_defects_all.csv, cke_release.xml, cm_release.csv
- **CM + CKE combined:** files_with_defects_all.csv, cke_release.xml, cm_release.csv

To provide a concrete example of a final dataset, Appendix 1 contains a sample of the CKE + CM data set. File names have been replaced with a numeric id, other data is unchanged. As described above, the dataset contains a file name, code metrics data, change metrics data, and a categorization of the files into defective and clean.

## 5.2 Implementing a software defect prediction framework

This Subchapter describes the tools used and created for software defect prediction in this thesis. First is the machine learning framework used to provide classification functionality in this thesis, which is the Waikato Environment for Knowledge Analysis (Weka) [34]. It is a Java-based collection of machine learning algorithms, which includes a UI tool and a Java framework for data analysis. The Java framework is used in this implementation for the classifier implementation. In addition to classification, Weka provides various tools for data loading and processing. Weka was chosen as the framework due to it being used and found suitable in several other studies [17, 24, 27].

Additionally, a customized tool named Seidr is created. Seidr uses the Weka framework to provide data combining, preprocessing and classification. Furthermore, Seidr is used for classifier performance measurements and for reporting results of the classification. The rest of this Subchapter provides a more detailed description of the functionality offered by Seidr.

The first feature of Seidr is data combining. This is used to create the eight previously defined final versions of the collected data sets for defect prediction. Data can be combined for a single version, or for multiple versions.

Second feature of Seidr is data preprocessing. This is used to implement the data preprocessing techniques introduced in Chapter 4. For Undersampling, Weka's SpreadSubsample class is used. Different ratios for the balance of majority and minority class can be provided, for example with value 2 the ratio of majority to minority class will be 2:1. Weka also supports CFS, which can be set as a data preprocessing option in Seidr. Additionally, Weka includes a cost-sensitive meta-classifier, which can also be set in Seidr as a data preprocessing option with a parameter for the weight of FN misclassification. The costs for TP and TN are 0 and cost for FP is 1.

The third feature of Seidr is providing to the classifiers in the Weka framework. Weka contains implementations of all five machine learning algorithms introduced in Chapter 4. These can be used with the selected data preprocessing options to train and evaluate a classifier. Data from a previous version is used as training data, and data from the target version is used as testing data, based on which all performance reporting is done. In addition, each classifier can be paired with a either Bagging or Boosting meta-classifier.

The fourth feature of Seidr is reporting performance and results of a classification in a desired format. This is an important feature due to the limited functionality of customized reporting in Weka. This functionality can be used both for estimating performance with training and testing data, and for making predictions.

## 5.3 Narrowing down classifier selection

As the number of classifiers selected for testing in this thesis is high, the selection of classifiers is narrowed down in this Chapter before a more detailed performance analysis is conducted in the next Chapter.

### 5.3.1 Defining initial configurations

Due to the number of configurations and different classifiers available, the initial goal is to narrow down which classifiers perform best. This is done using as minimal configurations as possible, while making the performance comparisons equal. The

chosen configurations are justified below.

The configurations which need to be taken into account when comparing performance between classifiers are data preprocessing, the chosen classifier, data sets over which the performance measures are gathered, and the eight different metric data sets. For a limited performance comparison, only a few configurations can be used. For this purpose, the final data sets are limited only to CM + CKE alpha data set. CM + CKE is chosen to be able to include both metric types, and to disregard the choice between aggregating or combining change metrics. Additionally, only two data preprocessors, Bagging and Boosting are considered. CFS is used for the Bayesian classifiers for which it is essential, and all available version data sets are used. Finally undersampling is used due to the imbalanced data set. An optimal value for undersampling ratio is chosen based on running each classifier on only a data set of one version, 6.1. The chosen values for undersampling ranged from 1 to 4, and no undersampling for BN classifier.

Lastly, performance metrics for comparing the classifiers need to be chosen. Using all of the possible measures is impractical, which is why only two measures are chosen, while the rest of the measures are used only for a final performance analysis and comparisons. The fist selected measure is AUC, which as discussed, is generally accepted as a good overall measure of classifier performance. The second measure is chosen to represent the use cases of the RELEX RM team, which would only consider the top results measured by confidence of the prediction list. For this purpose, the average of the PPV measure at three cut-off points is used. The three cutoff points are the values of prediction confidence at row 50, 100 and 200 of the predictions, where prediction results are sorted by the prediction confidence value. This essentially measures how many correctly predicted defective files generally are in the most confident prediction rows.

### 5.3.2 Initial performance comparison and results

The results for the initial performance comparison between the classifiers are represented in Figure 4. The x-axis represent the average PPV value, and y-axis represents the AUC value of the classifier. The results for each classifier per each of the target versions are plotted as the function of the two values. The best results would be at coordinates (1,1), which would represent a perfect classification, and worst at (0,0).

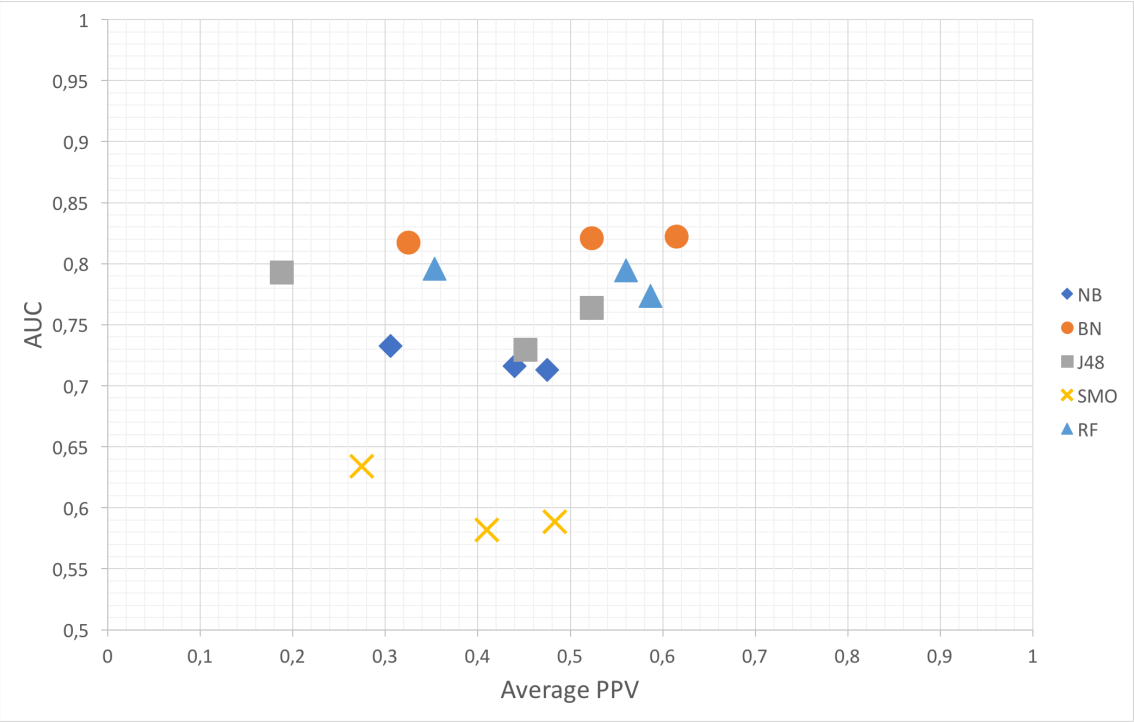


Figure 4: Initial results

The clear outlier in the results is the SMO classifier, which is a Weka implementation of SVM. The AUC score for SMO is noticeably lower than that of every other classifier, and the average PPV is in the lower-end of the results. Due to these factors, SMO is discarded from further analysis.

The next two in performance are the NB and J48 classifiers. While the AUC values for NB are not as clear outliers as for SMO, it nevertheless scores consistently lower compared to the rest of the classifiers, while having mostly worse average PPV values. The best quality of the NB classifier is that it scores fairly consistent values on both measures. J48 on the other hand achieves slightly better performance in terms of AUC than NB. However, the average PPV values are not very consistent, and overall mid-range at best. These two classifiers are thus also discarded from further analysis.

The remaining classifiers RF and BN score the best out of the five selected classifiers. The performance of these two classifiers is almost identical, with RF scoring slightly less in AUC but more consistently in average PPV. The result of these two classifiers performing best is not surprising, considering existing results from other studies [1]. Thus, the performance tuning and analysis will continue for BN and RF in the next Chapter.

Table 4: Initial scores

| | BN | RF | J48 | NB | SMO |
|---|---|---|---|---|---|
| **Score** | 1,31 | 1,29 | 1,15 | 1,13 | 0,99 |

The exact results are summarized in Table 4, where the score is calculated as the average of results of AUC and average PPV separately from the three different versions, and then the average of AUC and average PPV summed together. Here, a value of 2 would represent a perfect score.

# 6 Analysis

In this Chapter a more thorough analysis is conducted on the two selected classifiers, RF and BN. First, performance improvement optimizations are experimented on using previously discussed data preprocessing techniques and meta-classifiers. Then, the results are presented using classifier performance metrics, and finally, the usability of classification results is validated against the use cases defined in Chapter 2.

## 6.1 Improving results

In this Subchapter, performance improvements are experimented on BN and RF. The objective is to incrementally apply and evaluate techniques for improving results for both classifiers, with the goal of achieving the best performance for both classifiers.

Table 5: Initial results

| | Avg. PPV | AUC | Score |
|---|---|---|---|
| **BN** | 0,488 | 0,820 | 1,308 |
| **RF** | 0,5 | 0,789 | 1,289 |

The initial results on the two previously selected performance measures over three different version data sets can be seen in Table 5. The same performance measures will also be used to measure the performance after applying performance improvement techniques. Any performance improving techniques will be retained in the configuration in the tests following it.

### 6.1.1 Undersampling and cost sensitivity

The first performance improvement technique is to apply either Undersampling or Cost-sensitivity, and finding the best value for the selected technique. To achieve the initial results presented in the previous Subchapter, Undersampling with an initial best guess value was applied. For RF this was 3.0, and for BN no Undersampling. The comparison between Undersampling and cost-sensitivity which is done now thus also validates the correctness of the initial choices.

The measured values for different configurations for undersampling and Cost-sensitivity are plotted in Figure 5 for the BN classifier, and Figure 6 for the RF classifier. The circle-shaped dots are the values of Undersampling (US) and triangle-shaped dots are the values for the cost matrix (CM) of Cost-sensitivity. As before, x-axis represents the average PPV value and y-axis the AUC value.

For the BN classifier, the plot shows that Cost-sensitivity is the better technique both in terms of AUC and average PPV. The best choices for cost matrix values are between 5 and 10, where the range of the sum of AUC and average PPV is between 1,320-1,323, making the choices almost identical in terms of performance. The CM value 8 is chosen as the final value, as average PPV value is important for the use case of improving testing process, and the trade-off in terms of AUC value is minimal. Additionally, the sum of the two measures was highest for CM value 8. The exact values for average PPV and AUC achieved with CM value 8 are listen in Table 6. The performance measured by average PPV is better than with the initial setup, while the AUC value remains the same.

Table 6: Results after feature selection and cost sensitivity

| | Avg. PPV | AUC | Score |
|---|---|---|---|
| **BN** | 0,506 | 0,817 | 1,323 |
| **RF** | 0,509 | 0,788 | 1,297 |

For the RF classifier, Undersampling performs considerably better than Cost-sensitivity. Additionally, the average PPV value correlates with higher undersampling values. This is due to the classifier predicting less and less defective rows with an apparently higher correctness rate. However, the AUC value decreases slightly the more Undersampling is used. Additionally, while not a significant consideration in this case, processing time is also longer the higher undersampling values are. The value of 4.0 is chosen as the best trade-off between the two measures for Undersampling. The performance measures with Undersampling value 4.0 are listed in Table 6. As with the BN classifier, the AUC value remains the same compared to initial results, while PPV performs slightly better.

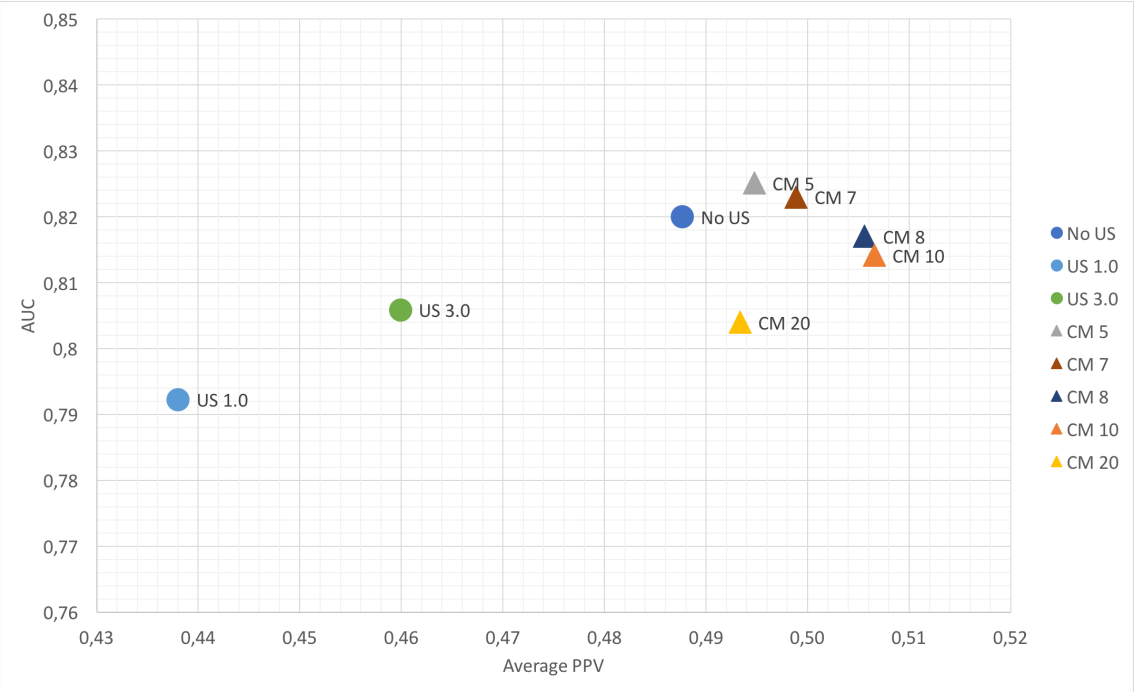


Figure 5: Cost Matrix and Undersampling for BN

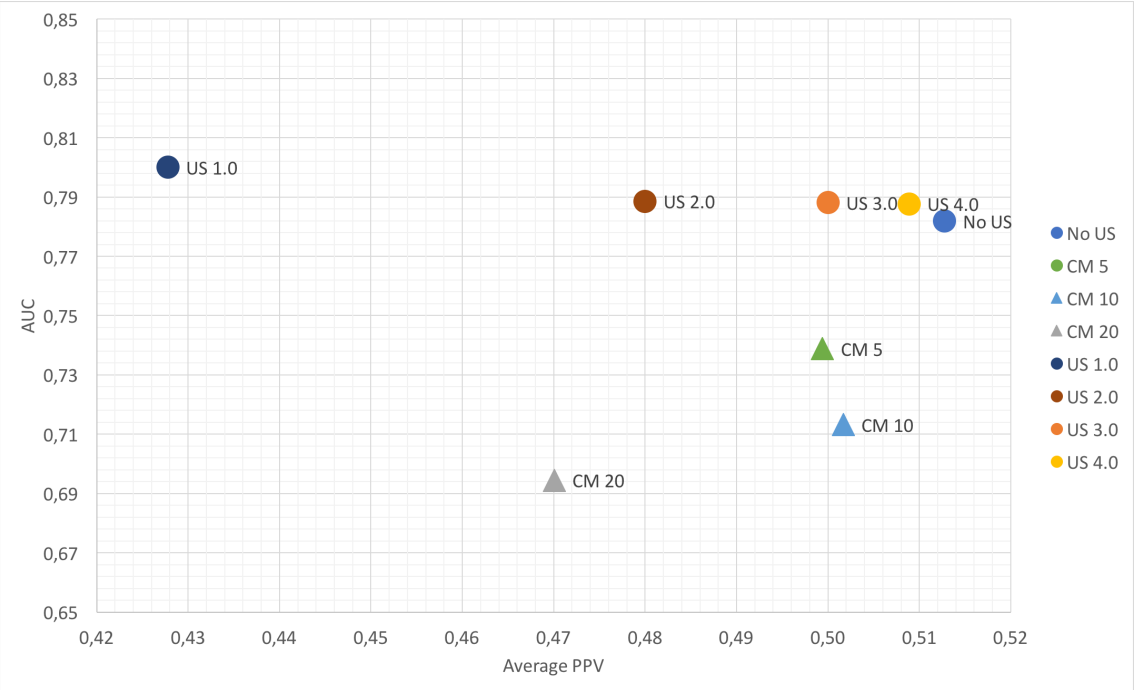


Figure 6: Cost Matrix and Undersampling for RF

### 6.1.2 Feature selection

Feature selection was also applied when calculating the initial results. Here, the selection to use Feature selection with the BN classifier, and no feature selection with the RF classifier, is validated. For this purpose, the RF classifier is tested with Feature selection and the BN classifier without. The results are shown in Table 7.

Table 7: Feature selection validation

| | Avg. PPV | AUC | Score |
|---|---|---|---|
| **BN without CFS** | 0,460 | 0,834 | 1,227 |
| **RF with CFS** | 0,462 | 0,837 | 1,300 |

The results provide no surprises, with both classifiers achieving lower performance values than with the initial setup. Thus the initial configurations for Feature selection are kept going forward.

### 6.1.3 Log filtering

Log filtering is applied similarly to the two classifiers as Undersampling and Cost-sensitivity. As every independent variable is numerical, the filter can be applied to every variable.

The results for both classifiers with Log filtering are seen in Table 8. Log filtering does not improve the results for either of the classifiers. For the RF classifier, the results are essentially unchanged, with the sum of the measured being only 0.2 smaller. The same applies for BN, with the exception that the loss in value comes mostly from PPV and not AUC. In conclusion, Log filtering does not impact the classification results considerably and thus will not be used in this case.

Table 8: Log filtering results

| | Avg. PPV | AUC | Score |
|---|---|---|---|
| **BN** | 0,482 | 0,820 | 1,300 |
| **RF** | 0,505 | 0,772 | 1,277 |

### 6.1.4 Data normalization

Data normalization is not required for either of the classifiers to perform well. For the RF classifier, the algorithm uses only absolute values to create the decision trees, thus normalization should not affect the performance. The BN classifier compares the features only with each other, meaning that Data normalization should leave the probabilities calculated for the classifier the same, despite the scale of the values.

This hypothesis is validated by running both test cases with both classifiers using normalized data. The results show that using normalized data has no effect on either of the classifiers, and thus it is not included in the configurations.

### 6.1.5 Deciding between data sets

Deciding between the data sets is essentially the question of whether to use code metrics, change metrics, or both. As discussed, there are in total eight data sets, three for predicting defects at alpha stage, and five for predicting defects at release. Before comparing performance with different data set configurations, a quick look is taken which files each type of data truly covers, and how many defects are already left out at this stage.

The CKJM metrics only consider Java class files, and thus leave out a considerable amount of defects. Approximately only 25% of defective files are Java class files, which makes the data set less valuable for comprehensive defect prediction. The change metrics on the other hand consider only files that have had changes made. Initial testing shows that the change metrics cover over 80% of all defective files, still leaving some files out, but considerably less than with CKJM only. The percentage improves only marginally when using combined data set of CKJM and change metrics, compared to only change metrics, making the two data sets more comparable.

The results of average PPV and AUC values are plotted for each data set, over the three versions. The results for the BN classifier are shown in Figure 7, and for the RF classifier in Figure 8. The data sets used for alpha defect prediction are plotted as triangles, and data sets used for release defect prediction as circles.

The most noticeable difference between data sets for both classifiers is the difference in performance of the CKE only data sets. The AUC values are considerably higher than other data sets for all CKE data sets, while the average PPV is generally

lower, with the exception of the alpha CKE metrics for RF. This is explained by the low amount of defects covered by the CKE set and the amount of files still being relatively high. Due to this, the ratio of defective files to total files is higher. The classifiers are able to classify effectively most of the clean files from the defective, which explains the high AUC value. However, due to the low amount of defective files the average PPV value is still generally lower than for the other data sets, since the rate of correctly detected defective files would need to be higher than it needs to be for other data sets to achieve same results in average PPV value. All in all, the performance for change metric data sets for both classifiers is better than the other data sets, but when considering the use cases, the performance is equal or worse, in addition to which the data set covers less files and defects. Due to these factors, the used data set will be picked out of the remaining data sets.

For the BN classifier, the unified data sets of CKJM and change metrics perform noticeably better for both alpha and release defect prediction than the rest of the data sets. The difference in performance is smaller for the release phase defect prediction, with the data sets using only change metrics performing slightly better in terms of average PPV value. However, due to the considerably higher AUC value of the unified data sets, these are selected for both alpha and release defect prediction. Of the two ways to unify change metric data sets, aggregation performed best.

For the RF classifier, the differences in AUC value between data sets were smaller. The best performing data sets were the same as for the BN classifier, although the difference between CM + CKE combined and CM + CKE aggregated was almost insignificant. However, the sum of AUC and average PPV was slightly lower for the combined change metrics. For similar reasons as BN, the same datasets are chosen for usage with RF.

### 6.1.6 Bagging and Boosting

In some situations the result of a classifier can be enhanced by the usage of a meta-classifier. The two meta-classifier techniques introduced, Bagging and Boosting, will be applied to the selected classifiers to determine if the results can be improved. The meta-classifiers will be tested on both of the defect prediction cases.

The RF classifier itself is an ensemble classifier, which combines results from decision trees. Thus it is not beneficial to further pair the RF classifier with another meta-

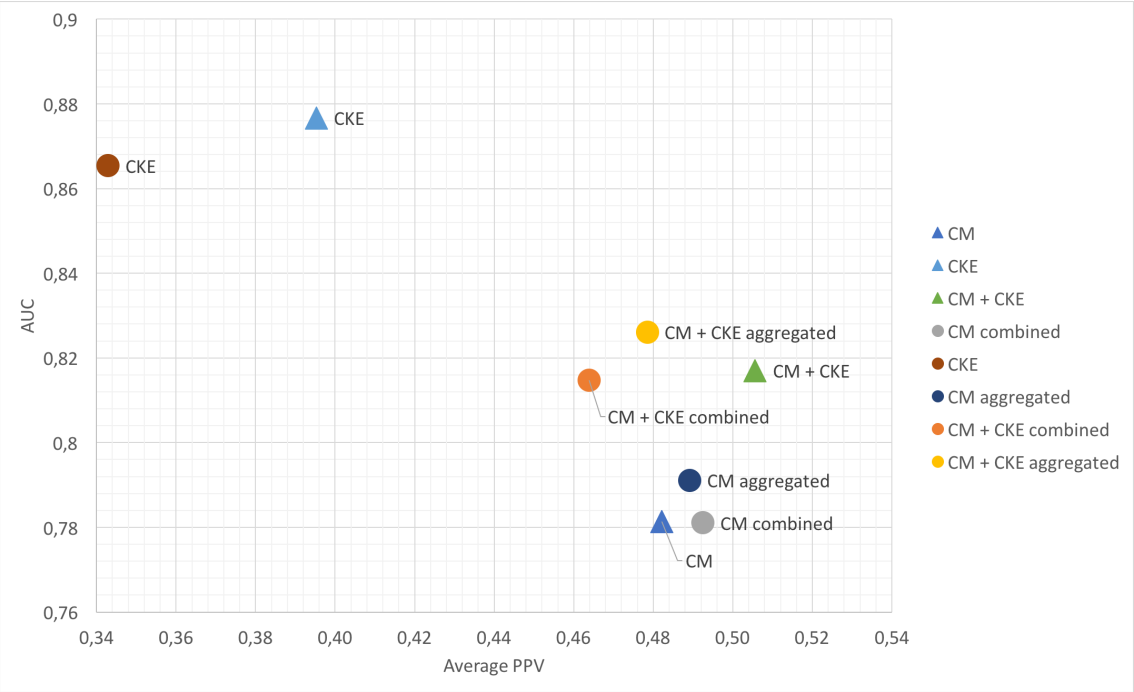


Figure 7: Data set comparison for BN

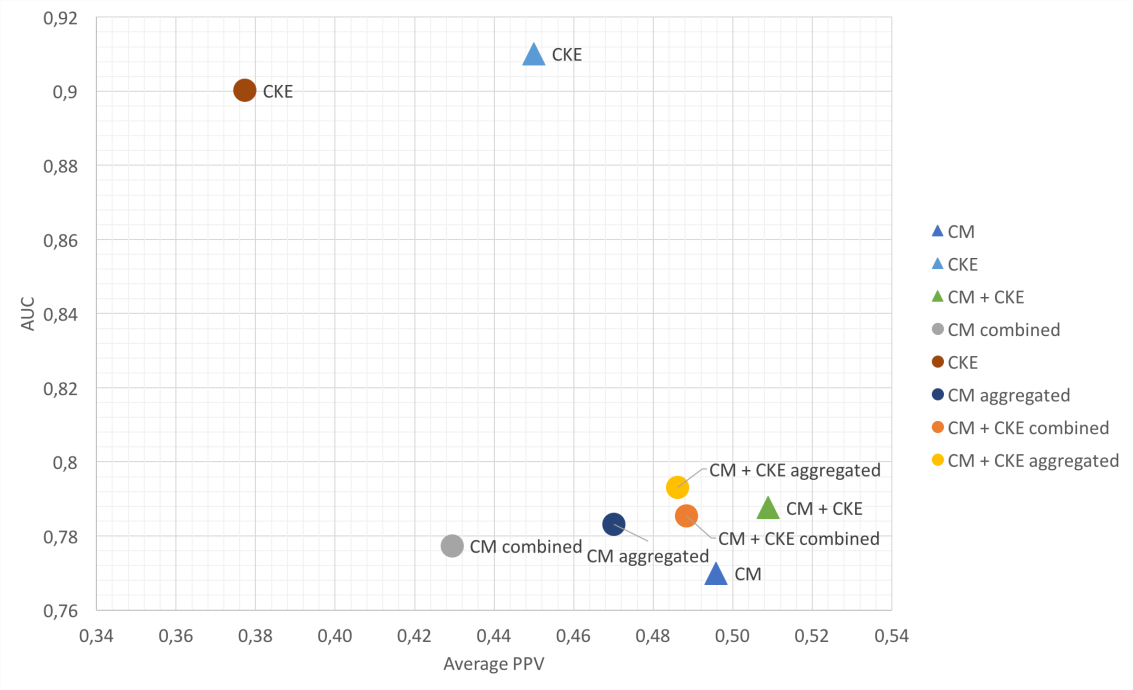


Figure 8: Data set comparison for RF

classification technique, and will not be selected here for testing with Bagging or Boosting.

Table 9: BN with meta-classifiers

| | Avg. PPV | AUC | Score |
|---|---|---|---|
| **Adaboost / alpha** | 0,257 | 0,784 | 1,041 |
| **Adaboost / release** | 0,233 | 0,777 | 1,010 |
| **Bagging / release** | 0,479 | 0,827 | 1,305 |
| **Bagging / release** | 0,459 | 0,831 | 1,290 |

The results for BN with Bagging and Boosting can be seen in Table 9. It shows results for both defect prediction cases, with both meta-classification techniques. The Adaboost technique does not pair well with BN in this case, with results being considerably lower in both AUC and especially average PPV. The Bagging technique performs considerably better than Adaboost. The average PPV value is slightly lower, while AUC value is slightly higher. The sum of the two measures is still however lower than for BN without Bagging. Due to this, in addition to the lower average PPV value, Bagging is not used in this case.

## 6.2 Results for the final configurations

In this Chapter the final configurations are evaluated and compared to studies with similar setups using the commonly used classifier performance measures presented in Chapter 4. The performance measures will be reported for cut-off point value 0.5 to be able to better compare the results to results achieved in other studies. Additionally, the results of Feature selection will be presented, to better understand which features perform best at defect prediction in this case. Finally, a summary of the selected configurations, and performance results, will be provided.

### 6.2.1 Feature selection results

The selected features by the classifiers provide insight into the classifiers performance, and allow to compare the selections to other studies. As the RF classifier

does not use CFS and does not provide selection analysis, the CFS selection of BN will be used.

Over three data sets for two defect prediction cases, the most selected features with five or six inclusions out of six classifications were: Authors, Avg. Changeset, Codechurn, LCOM, LOC added, LOC removed, Revisions and Weighted age. The results are interesting since only one code metric, LCOM, has frequently been included for classification. Furthermore, it is the LCOM metric, instead of LCOM3. One explanation for it would be a high correlation between the two measures. In any case, this finding would imply that the lack of cohesion measured by the metric is a good predictor for defect proneness of a file. The second most frequent code metric with 3 inclusions is RFC, which interestingly is also a CK metric, solidifying the validity of CK metrics as the most used metric set.

On selected change metrics, the first observation is that metrics with different aggregation method were very rarely selected twice, for example average LOC added and total LOC added. As CFS selects metrics based on correlation with other metrics, this means that the aggregated metrics correlate with the base metric, and that aggregated metrics might not be necessary in this case. Perhaps the most interesting often selected change metric is the Authors metric, which implies that the more different authors have made changes the more defect prone the file is. While this initially seems to imply that the number of authors should be kept to minimum, it also likely correlates with the number of changes that have been made.

### 6.2.2 Performance results and comparison to other studies

To compare the overall classification performance to other similar studies, the performance measures in general use are applied. The values will be calculated for a cut-off point of 0.5, as that is the value used in most of the studies. For the sake of brevity, the results are calculated for three data sets for each of the two cases together. The results of the calculation are plotted as a min-max-average plot. The performance measures can be seen in Figure 10 for the BN classifier, and Figure 9 for the RF classifier. When looking at the results it should be kept in mind that most of the measures correlate to a degree with AUC, which was not the only or necessarily the main focus when optimizing classifier performance due to the selected use cases.

Compared to each other, the classifiers have some notable differences in performance. Firstly, the RF classifier appears to perform more consistently in most performance

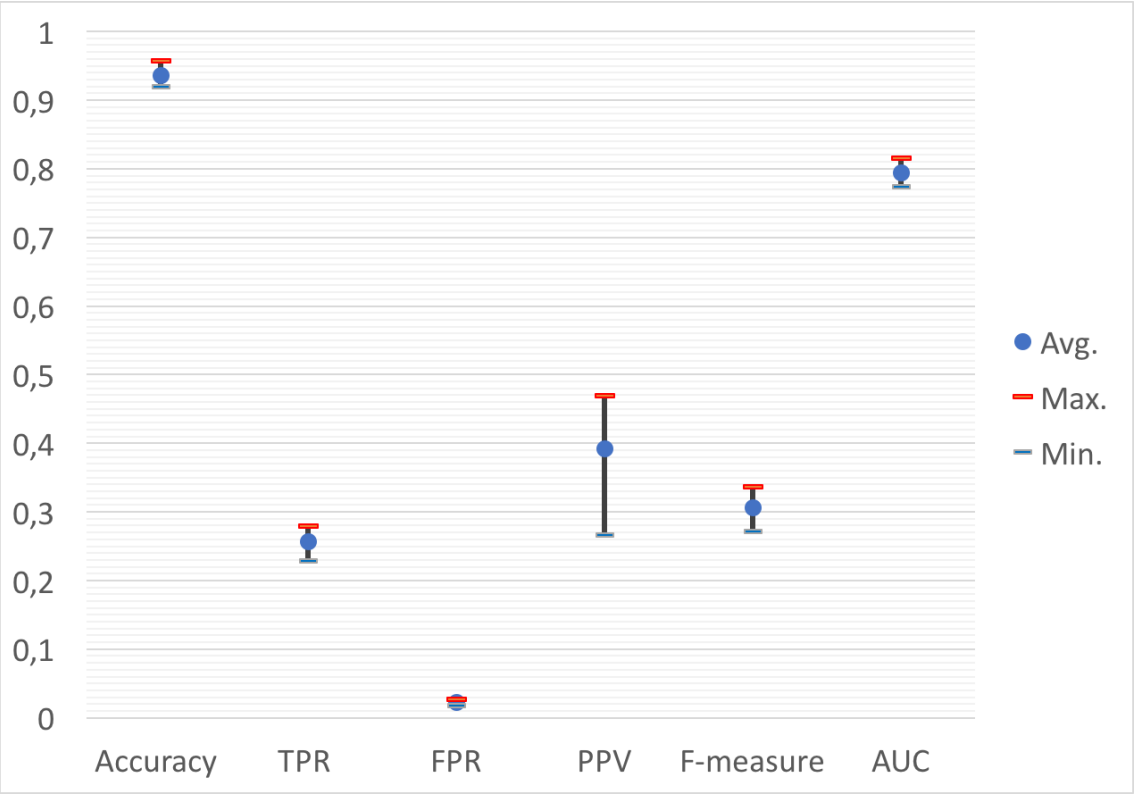


Figure 9: Performance measures for RF

measures compared to BN, with the exception of PPV., where BN while achieving lower values does it more consistently. The second difference is the trade-off between TPR and FPR. Compared to the BN classifier, RF has lower FPR values, but the TPR values are accordingly considerably lower. As keeping FPR low is of critical importance in defect prediction, due to imbalanced data, it looks like RF performs better in this regard. However, both BN and RF achieve similar values in terms of AUC, and the AUC measures how good trade-offs can be achieved between TPR and FPR. This means that the different trade-off made for the classifiers in TPR and FPR, while different, is nevertheless similar from the point of view of performance measurement.

Due to using both change and code metrics, the performance measures are compared to studies which also use both metrics. The study conducted by Choudary et al. is the most similar study conducted to the software defect prediction implementation in this thesis [15]. In the study, both code and change metrics were used, and the selected metrics were largely the same as in this thesis. The RF classifier was also used, but not BN or NB which would have been comparable. Thus only RF classifier results are compared. The results in Choudary et al. study when training with a

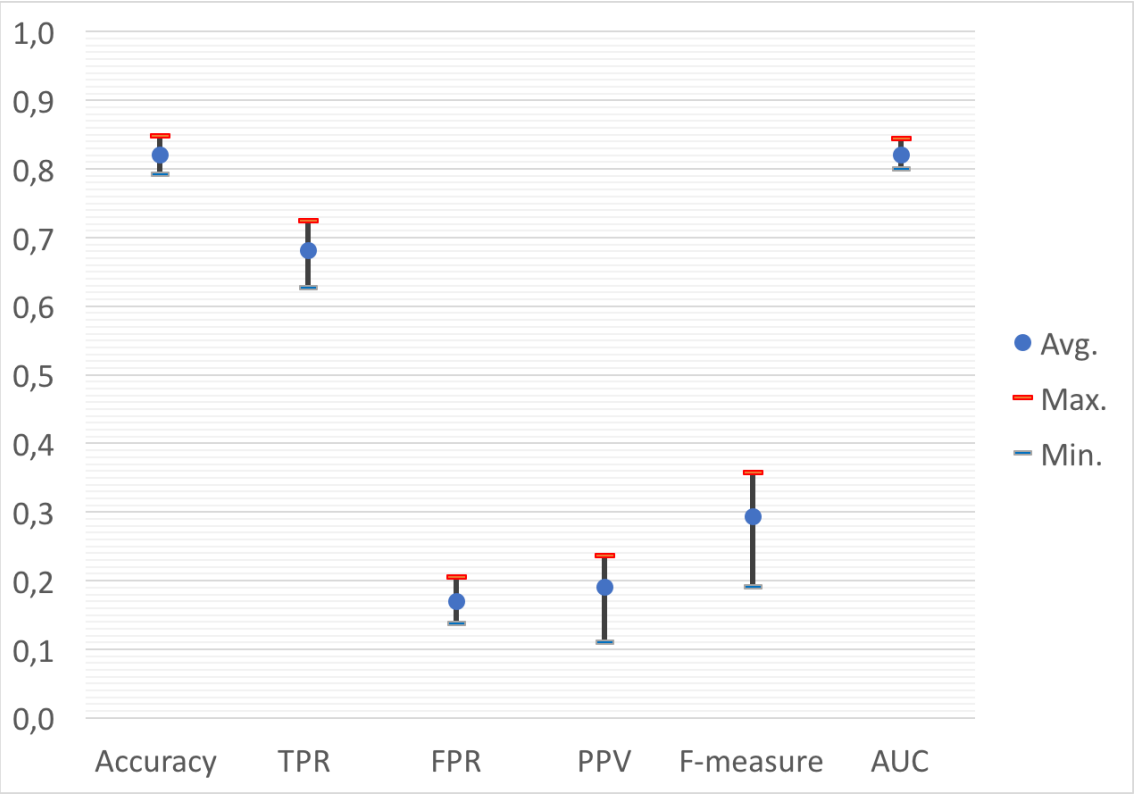


Figure 10: Performance measures for BN

previous version and testing with current were similar to the results achieved here with the RF classifier. Most notable difference was that the maximum PPV value achieved was higher in some cases. However, TPR and F-measure were generally lower. The differences are arguably surprisingly small despite using different data, and small difference in configuration. Similarly to this thesis, Choudary et al. recommend that change metrics and code metrics should be used together to achieve best performance.

The study conducted by Moser et al. also used code and change metrics [8], but the results of the study are only somewhat comparable, as only accuracy, TP and FP values are reported. Additionally, the used data set size was smaller, and the used classifiers in the study includes NB, but neither BN or RF. Despite the differences, accuracy values achieved in the study were comparable to the results of BN classifier in this thesis, further validating the results as acceptable compared to other studies.

### 6.2.3 Final results and discussion

To conclude the configuration and performance evaluation, a summary of the results is presented here. The best results measured by average PPV and AUC can be found in Table 10. Additionally overall performance measures can be found in Figure 9 and Figure 10.

Table 10: Final results

| | Avg. PPV | AUC | Score |
|---|---|---|---|
| **BN / alpha** | 0,506 | 0,817 | 1,323 |
| **RF / alpha** | 0,509 | 0,788 | 1,297 |
| **BN / release** | 0,479 | 0,826 | 1,305 |
| **RF / release** | 0,509 | 0,788 | 1,297 |

The configuration used for the RF classifier is Weka's default parameters, in addition to undersampling with a four to one split between clean and defective files. For the BN classifier Weka's default configuration was also used. Additionally, Cost-sensitive meta-classifier was used with a cost value 8 for misclassification of defective files, and Weka's implementation of CFS was applied to data before classification.

Lastly, the choice of cut-off point selection is discussed here. While no specific cut-off point is determined, the cut-off points that were used were defined as the the confidence value of the prediction at row 50, 100 and 200 when the results were sorted by the confidence value. In general, the confidence value at 200 rows for the RF classifier was approximately 0.5, which is the default. The confidence values at 100 and 50 were slightly higher. For the BN classifier however, then confidence values were very high, with the value being over 0.999 even for confidence value at 200 rows. Based on these observations, the RF classifier could be used without any additional cut-off point configuration in this case, while the cut-off point for BN should be considered case-by case.

## 6.3 Use case validation

In this Subchapter, the prediction results are evaluated against the use cases defined by the research questions. The evaluation is conducted in cooperation with RELEX

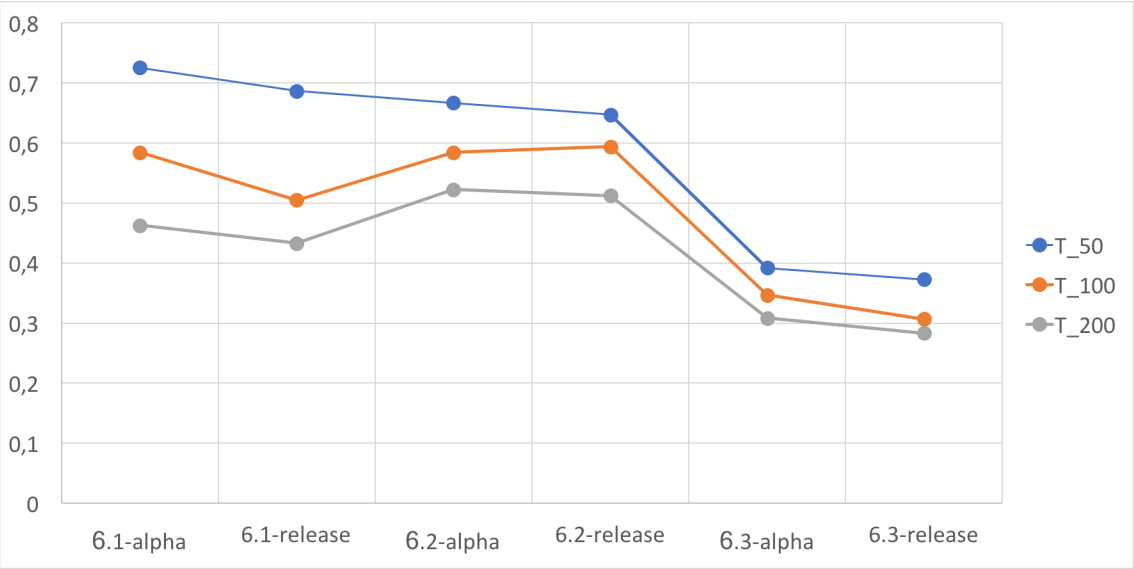


Figure 11: PPV values for RF classifier for different RELEX software versions

RM team. The first RQ is validated using a more thorough analysis on PPV values, and the potential for selecting a suitable release time for the second RQ is investigated. Additionally, an overall result quality and usefulness validation is conducted by the RM team.

### 6.3.1 Validating usability for testing process improvement

As discussed before, the RM team would only use the prediction results with the highest confidence value and thus PPV is the best performance measure to estimate the usefulness of the prediction for this use case. As such, a closer look is taken into the achieved PPV values for both classifiers.

The PPV values for each version for RF are shown in Figure 11. Three lines are plotted for PPV value at 50, 100 and 200 rows respectively. The first observation from the Figure is that in general, prediction accuracy measured by PPV does not differ between alpha and release versions. This is a positive quality, as this shows that the prediction is suitable for both alpha and release phase. Moreover, the difference between PPV value at different number of rows is relatively consistent, with some exceptions such as 6.1 alpha and release. In general the PPV values show that the prediction performance is acceptable for the desired use case. This is especially true when considering PPV at 50 rows, which is on average 0.7 for software versions 6.1 and 6.2. However, for software version 6.3 the values are considerably lower.

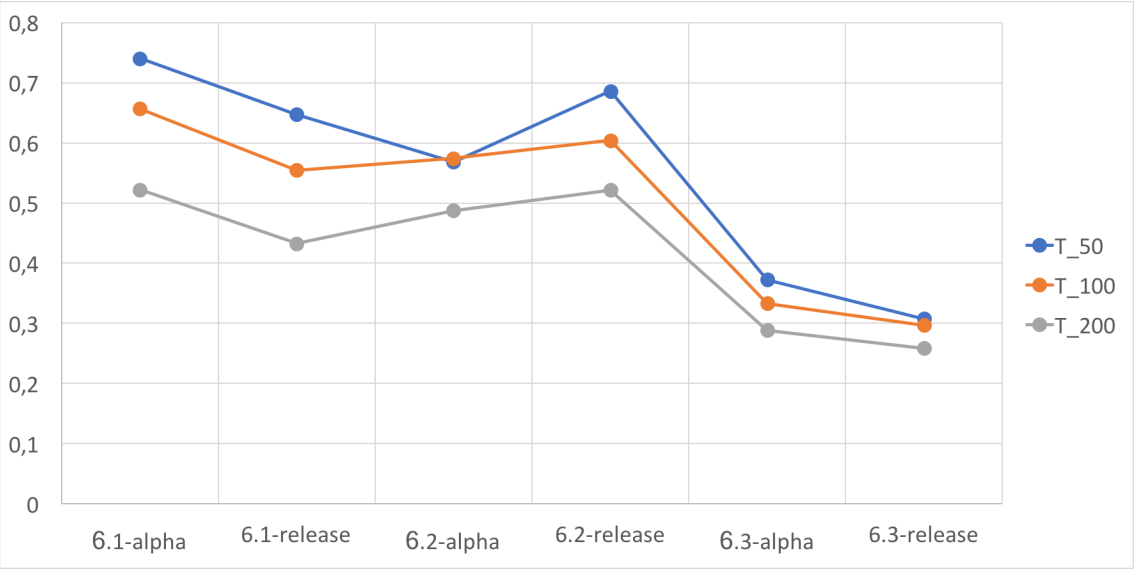


Figure 12: PPV values for BN classifier for different RELEX software versions

This reduces the usability as the reliability of the prediction cannot be assumed to be consistent. The PPV value of under 0.4 is still usable for the use case, but its usefulness is considerably lower.

The PPV values for the BN classifier are plotted in Figure 12. The results are similar to the PPV values achieved with RF. However, the variance in PPV values for versions 6.1 and 6.2 at 50 rows is higher than for RF, even though the best results are slightly higher. Similarly to RF, the PPV values for BN are considerably lower for version 6.3. This suggests that the cause for the lower PPV values is in the data, with the most likely explanation being that the testing data contained less defects than for other versions, while training data contained more. The trend in the number of defective files in test data can be seen in Figure 13.

### 6.3.2 Validating usability for version defectiveness estimation

The number of rows predicted as defective optimally indicates the overall defectiveness of a given software version. This is related to the RQ2, as the more defects there are in the system, the worse time it is to release the version. This hypothesis is validated by comparing the number of rows predicted defective to the amount of defects that was actually in the version.

The number of rows predicted defective are plotted in Figure 13 for both classifiers, in addition to the number of defects in test data, i.e. the version in question. The

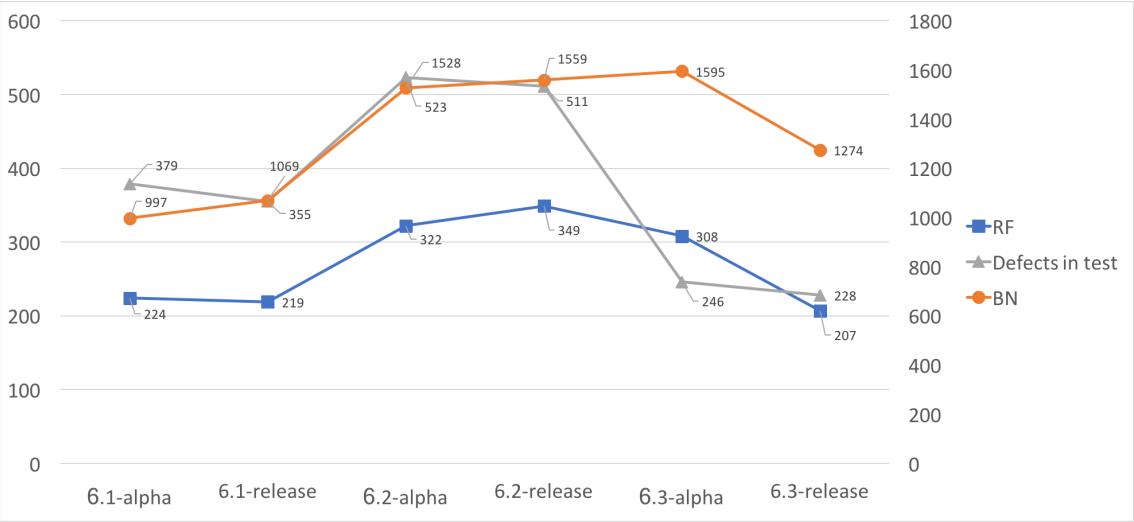


Figure 13: Number of result rows predicted defective

left-side y-axis contains value range for RF and the number of defects in test data, and right-side y-axis contains the value range for BN. A cut-off point value of 0.5 is used to calculate the amount of rows predicted defective, as a fixed cut-off point recommendation was not defined for either of the classifiers, and a set number of rows which was used for PPV cannot be used in this case.

For versions 6.1-alpha to 6.2-release, the number of defects in the version correlates strongly with the number of rows predicted defective. However, for version 6.3-release and especially for 6.3-alpha, while the correlation is still present, it is considerably lower. This is likely due to the same reason which caused a similar drop in PPV value for version 6.3, which is the large amount of defects in training data, but considerably lower amount of defects in testing data.

The second observation from Figure 13 is the considerable difference between number of rows predicted defective between the two classifiers. This is due to the cut-off point value recommendation being considerably higher for BN than 0.5. Despite this, the correlation between BN and defects in test is effectively the same as for RF.

### 6.3.3 Validating prediction usefulness

The quality and usefulness of the results was validated by RELEX RM team. The process for validation was to check the top results from the predictions made by both classifiers, and check what kinds of bug fixes had been made to those files.

In general, the quality of the predictions was found to be good. The rows which were correctly predicted as being defective contained relevant bug fixes that were implemented in later versions. This further validates the usefulness of the prediction rows with highest confidence values. The difference between the two classifiers in terms of predicted files was minimal, with both ranking same files to a large extent in the top prediction confidence rows.

Despite only small differences in results between BN and RF, the RF classifier was found to perform overall better. This was mainly due to ease of configuration and easier cut-off point selection. The latter also affected performance for the use case described in RQ2, which further supports the choice of RF. However, due to the differences being relatively small between the classifiers in performance, it is recommended to further monitor both classifiers, and further experiment with different configurations.

All in all, the current results are promising for the two defect prediction cases presented. Currently the main concern regarding performance is the drop in PPV values for software version 6.3. Validation of the framework will continue at RELEX with future software versions, which will provide more data and thus a better estimate of the average defect prediction performance. If the results prove to be consistently acceptable, the framework can be used in practice for testing purposes and deciding release times of the software.

# 7 Summary

In this thesis a software defect prediction framework was implemented to assist in targeting testing resources and selecting optimal release times. The implementation used software metrics and machine learning techniques to produce a list of files in the RELEX software which were most likely to contain defects. The performance of the implemented software defect prediction framework was validated by common classifier performance metrics, and manual use case validation.

Based on analysis of the defect prediction performance, the RQ1 can be answered tentatively positively. The prediction performance measured by PPV value for top 50 to 200 rows was overall high enough, even for version 6.3, that it could be used as a tool for concentrating manual testing resources to those parts of the software. However, further testing and monitoring of the results is still required to determine if the reliability of the implemented software defect prediction framework is high enough for practical use.

Similarly, the answer to RQ2 is tentatively positive. It was shown that the number of result rows predicted defective correlates with the number of defects in the software, measured by the amount of fixed defects in the next version. Again, software version 6.3 showed worse results in this regard than the two previous software versions, which reduced the reliability of the result. Thus further testing and monitoring of the results is required with future versions to better determine if the framework provides acceptably reliable results.

In addition to continuing monitoring the performance in future versions, several other measures can be taken in the future to further improve defect prediction performance. Firstly, more software metrics can be tried and validated in the defect prediction, for example Choudary's change metrics. Additionally, performance evaluation can be performed on other classifiers and configurations for the classifiers which were already experimented on can be further validated and optimized based on future results.

Additionally, going forward with performance validation, it should be observed whether the drop in performance seen in version 6.3 is an anomaly or does it correlate with large differences in defect counts in the training data, i.e. the previous version. Currently, the performance drop does lower the confidence of the results and thus if the drops are common then it will affect the usability of the predictions considerably. On the other hand, the effect can somewhat be mitigated if it is taken

into account when looking at the predictions whether the last version contained an abnormally large amount of defect fixes. Alternatively, if the performance of RF and BN stabilizes on the lower performance levels, then other classifier options should be looked at.

The final improvement suggestion is to create a complementary tool which would assist in the usage of prediction results. The tool would improve the usage of prediction list for testing by linking the files which were predicted as defective back to the issues in Redmine. This would provide context for which changes could have caused the defect in the file, and more specifically where to look for it in the file.

# References


1 R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

2 S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

3 M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pp. 242–249, IEEE, 1999.

4 J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

5 S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

6 E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of Systems and Software*, vol. 83, no. 1, pp. 2–17, 2010.

7 T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

8 R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*, pp. 181–190, ACM, 2008.

9 R. Martin, "Oo design quality metrics," *An analysis of dependencies*, vol. 12, pp. 151–170, 1994.

10 D. R. Kumar and G. Kaur, "Comparing complexity in accordance with object oriented metrics," *International Journal of Computer Applications*, vol. 15, no. 8, pp. 42–45, 2011.

11 W. Li, "Another metric suite for object-oriented programming," *Journal of Systems and Software*, vol. 44, no. 2, pp. 155–162, 1998.

12 L. Badri and M. Badri, "A new class cohesion criterion: An empirical study on several systems," *Proceedings of QAOOSE*, vol. 3, 2003.

13 T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pp. 107–116, IEEE, 2010.

14 D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

15 G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault prediction," *Computers & Electrical Engineering*, vol. 67, pp. 15–24, 2018.

16 C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Information Sciences*, vol. 179, no. 8, pp. 1040–1058, 2009.

17 R. Malhotra and Y. Singh, "On the applicability of machine learning techniques for object oriented software fault prediction," *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 24–37, 2011.

18 R. Malhotra and A. Jain, "Fault prediction using statistical and machine learning methods for improving software quality," *Journal of Information Processing Systems*, vol. 8, no. 2, pp. 241–262, 2012.

19 L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

20 I. Rish, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, pp. 41–46, IBM, 2001.

21 N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers," *Machine learning*, vol. 29, no. 2-3, pp. 131–163, 1997.

22 T. S. Korting, "C4. 5 algorithm and multivariate decision trees," *Image Processing Division, National Institute for Space Research–INPE Sao Jose dos Campos–SP, Brazil*, 2006.

23 D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Software defect prediction using static code metrics underestimates defect-proneness," in *Neural*

*Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1–7, IEEE, 2010.

24 K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

25 Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A general software defect-proneness prediction framework," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 356–370, 2011.

26 M. A. Hall, "Correlation-based feature selection of discrete and numeric class machine learning," 2000.

27 S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 434–443, 2013.

28 T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proceedings of the 4th international workshop on Predictor models in software engineering*, pp. 47–54, ACM, 2008.

29 G. M. Weiss, K. McCarthy, and B. Zabar, "Cost-sensitive learning vs. sampling: Which is best for handling unbalanced classes with unequal error costs?," *DMIN*, vol. 7, pp. 35–41, 2007.

30 J. R. Quinlan *et al.*, "Bagging, boosting, and c4. 5," in *AAAI/IAAI, Vol. 1*, pp. 725–730, 1996.

31 T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software engineering*, vol. 31, no. 10, pp. 897–910, 2005.

32 M. Jureczko and D. Spinellis, *Using Object-Oriented Design Metrics to Predict Software Defects*, vol. Models and Methodology of System Dependability of *Monographs of System Dependability*, pp. 69–81. Wroclaw, Poland: Oficyna Wydawnicza Politechniki Wroclawskiej, 2010.

33 M. Aniche, "Change metrics." `https://github.com/mauricioaniche/change-metrics`, 2015.

34 I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

# Appendix 1. Final data CM + CKE

| file | wmc | dit | noc | cbo | rfc | lcom | ca | ce | npm | lcom3 | loc | dam | moa | mfa | cam | ic | cbm | amc | cc | revisions | authors | locAdded | locRemoved | maxLocAdded |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 72,00 | 32,00 | 1562,00 | 1362,00 | 91,00 |
| 2 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 2,00 | 2,00 | 37,00 | 1,00 | 36,00 |
| 3 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 2,00 | 1,00 | 6,00 | 6,00 | 4,00 |
| 4 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 2,00 | 2,00 | 24,00 | 2,00 | 23,00 |
| 5 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 53,00 | 0,00 | 53,00 |
| 6 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 2,00 | 2,00 | 1,00 | 1,00 | 1,00 |
| 7 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 4,00 | 4,00 | 34,00 | 8,00 | 20,00 |
| 8 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 20,00 | 0,00 | 20,00 |
| 9 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 4,00 | 2,00 | 79,00 | 5,00 | 45,00 |
| 10 | 6 | 0 | 0 | 14 | 16 | 11 | 2 | 13 | 3 | 0,60 | 63,00 | 1,00 | 1,00 | 0,00 | 0,33 | 0,00 | 0,00 | 9,17 | 1,00 | 2,00 | 1,00 | 93,00 | 6,00 | 91,00 |
| 11 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 2,00 | 1,00 | 88,00 | 88,00 | 60,00 |
| 12 | 0 | 1 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 2,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 | ? | ? | ? | ? | ? |
| 13 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 5,00 | 0,00 | 5,00 |
| 14 | 14 | 1 | 0 | 34 | 21 | 91 | 28 | 14 | 14 | 2,00 | 52,00 | 0,00 | 0,00 | 0,00 | 0,15 | 0,00 | 0,00 | 2,71 | 1,00 | 3,00 | 1,00 | 841,00 | 78,00 | 595,00 |
| 15 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 371,00 | 0,00 | 371,00 |
| 16 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 3,00 | 3,00 | 9,00 | 7,00 | 5,00 |
| 17 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 61,00 | 51,00 | 902,00 | 72,00 | 72,00 |
| 18 | 2 | 0 | 0 | 5 | 3 | 1 | 1 | 5 | 2 | 2,00 | 29,00 | 0,00 | 0,00 | 0,00 | 0,58 | 0,00 | 0,00 | 13,50 | 1,00 | ? | ? | ? | ? | ? |
| 19 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 3,00 | 3,00 | 3,00 |
| 20 | 43 | 1 | 0 | 33 | 188 | 701 | 5 | 32 | 8 | 0,86 | 1556,00 | 0,83 | 6,00 | 0,00 | 0,12 | 0,00 | 0,00 | 34,91 | 3,00 | 15,00 | 10,00 | 549,00 | 460,00 | 217,00 |
| 21 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 0,00 | 0,00 | 0,00 |
| 22 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 20,00 | 9,00 | 202,00 | 82,00 | 59,00 |
| 23 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 0,00 | 0,00 | 0,00 |
| 24 | 13 | 1 | 0 | 10 | 31 | 30 | 1 | 10 | 9 | 0,79 | 184,00 | 1,00 | 1,00 | 0,00 | 0,18 | 0,00 | 0,00 | 12,69 | 1,00 | ? | ? | ? | ? | ? |
| 25 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 2,00 | 2,00 | 1,00 | 1,00 | 1,00 |
| 26 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 0,00 | 0,00 | 0,00 |
| 27 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 2,00 | 1,00 | 42,00 | 4,00 | 38,00 |
| 28 | 109 | 0 | 0 | 59 | 160 | 5886 | 40 | 21 | 109 | 1,01 | 828,00 | 1,00 | 0,00 | 0,00 | 0,08 | 0,00 | 0,00 | 6,59 | 1,00 | 131,00 | 71,00 | 1278,00 | 490,00 | 22,00 |
| 29 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 2,00 | 2,00 | 45,00 | 5,00 | 45,00 |
| 30 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 0,00 | 0,00 | 0,00 |
| 31 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 14,00 | 0,00 | 14,00 |
| 32 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 10,00 | 7,00 | 43,00 | 19,00 | 16,00 |
| 33 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 237,00 | 0,00 | 237,00 |
| 34 | 18 | 1 | 0 | 22 | 41 | 87 | 14 | 8 | 15 | 0,42 | 405,00 | 1,00 | 1,00 | 0,00 | 0,38 | 0,00 | 0,00 | 21,11 | 2,00 | 2,00 | 2,00 | 8,00 | 7,00 | 5,00 |
| 35 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 12,00 | 6,00 | 162,00 | 45,00 | 122,00 |
| 36 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 5,00 | 4,00 | 107,00 | 80,00 | 91,00 |
| 37 | 10 | 0 | 0 | 6 | 17 | 37 | 0 | 6 | 4 | 0,72 | 77,00 | 1,00 | 1,00 | 0,00 | 0,45 | 0,00 | 0,00 | 6,50 | 1,00 | 1,00 | 1,00 | 52,00 | 0,00 | 52,00 |
| 38 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 0,00 | 0,00 | 0,00 |
| 39 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 4,00 | 4,00 | 71,00 | 70,00 | 51,00 |
| 40 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 3,00 | 3,00 | 11,00 | 15,00 | 8,00 |
| 41 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 3,00 | 2,00 | 208,00 | 210,00 | 156,00 |
| 42 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 1,00 | 0,00 | 1,00 |
| 43 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 0,00 | 0,00 | 0,00 |
| 44 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 1,00 | 1,00 | 1,00 | 0,00 | 1,00 |
| 45 | 2 | 0 | 0 | 7 | 4 | 1 | 2 | 7 | 0 | 1,00 | 24,00 | 0,00 | 3,00 | 0,00 | 0,58 | 0,00 | 0,00 | 9,50 | 1,00 | ? | ? | ? | ? | ? |
| 46 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 4,00 | 3,00 | 5,00 | 3,00 | 2,00 |
| 47 | 16 | 0 | 0 | 8 | 32 | 16 | 2 | 8 | 12 | 0,27 | 241,00 | 1,00 | 1,00 | 0,00 | 0,24 | 0,00 | 0,00 | 13,88 | 1,00 | ? | ? | ? | ? | ? |
| 48 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 7,00 | 4,00 | 74,00 | 19,00 | 30,00 |
| 49 | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | 4,00 | 4,00 | 36,00 | 37,00 | 28,00 |

| maxLocRemoved | avgLocAdded | avgLogRemoved | codeChurn | maxChangeset | avgChangeset | weeks | defective_class |
|---|---|---|---|---|---|---|---|
| 85,00 | 0,00 | 0,00 | 2924,00 | 916,00 | 0,00 | 47,00 | defective |
| 1,00 | 18,50 | 0,50 | 38,00 | 92,00 | 69,00 | 32,00 | clean |
| 4,00 | 3,00 | 3,00 | 12,00 | 276,00 | 258,50 | 9,00 | defective |
| 2,00 | 12,00 | 1,00 | 26,00 | 742,00 | 371,50 | 6,00 | clean |
| 0,00 | 53,00 | 0,00 | 53,00 | 194,00 | 194,00 | 0,00 | clean |
| 1,00 | 0,50 | 0,50 | 2,00 | 916,00 | 458,50 | 4,00 | clean |
| 7,00 | 8,50 | 2,00 | 42,00 | 916,00 | 434,75 | 38,00 | clean |
| 0,00 | 20,00 | 0,00 | 20,00 | 17,00 | 17,00 | 0,00 | clean |
| 3,00 | 19,75 | 1,25 | 84,00 | 916,00 | 452,50 | 31,00 | clean |
| 6,00 | 46,50 | 3,00 | 99,00 | 9,00 | 7,00 | 17,00 | clean |
| 60,00 | 44,00 | 44,00 | 176,00 | 276,00 | 258,50 | 9,00 | defective |
| ? | ? | ? | ? | ? | ? | ? | clean |
| 0,00 | 5,00 | 0,00 | 5,00 | 2,00 | 2,00 | 0,00 | clean |
| 76,00 | 280,33 | 26,00 | 919,00 | 43,00 | 34,67 | 23,00 | clean |
| 0,00 | 371,00 | 0,00 | 371,00 | 383,00 | 383,00 | 0,00 | clean |
| 5,00 | 3,00 | 2,33 | 16,00 | 916,00 | 318,67 | 5,00 | clean |
| 2,00 | 0,00 | 0,00 | 974,00 | 92,00 | 0,00 | 35,00 | clean |
| ? | ? | ? | ? | ? | ? | ? | clean |
| 3,00 | 3,00 | 3,00 | 6,00 | 17,00 | 17,00 | 0,00 | clean |
| 196,00 | 36,60 | 30,67 | 1009,00 | 659,00 | 64,20 | 44,00 | defective |
| 0,00 | 0,00 | 0,00 | 0,00 | 916,00 | 916,00 | 0,00 | clean |
| 24,00 | 10,10 | 4,10 | 284,00 | 916,00 | 62,55 | 47,00 | defective |
| 0,00 | 0,00 | 0,00 | 0,00 | 214,00 | 214,00 | 0,00 | clean |
| ? | ? | ? | ? | ? | ? | ? | clean |
| 1,00 | 0,50 | 0,50 | 2,00 | 916,00 | 649,50 | 13,00 | clean |
| 0,00 | 0,00 | 0,00 | 0,00 | 214,00 | 214,00 | 0,00 | clean |
| 4,00 | 21,00 | 2,00 | 46,00 | 742,00 | 468,00 | 23,00 | clean |
| 13,00 | 0,00 | 0,00 | 1768,00 | 181,00 | 0,00 | 50,00 | defective |
| 5,00 | 22,50 | 2,50 | 50,00 | 916,00 | 649,50 | 13,00 | clean |
| 0,00 | 0,00 | 0,00 | 0,00 | 916,00 | 916,00 | 0,00 | clean |
| 0,00 | 14,00 | 0,00 | 14,00 | 1,00 | 1,00 | 0,00 | clean |
| 10,00 | 4,30 | 1,90 | 62,00 | 614,00 | 91,80 | 43,00 | clean |
| 0,00 | 237,00 | 0,00 | 237,00 | 90,00 | 90,00 | 0,00 | clean |
| 4,00 | 4,00 | 3,50 | 15,00 | 659,00 | 368,00 | 22,00 | clean |
| 19,00 | 13,50 | 3,75 | 207,00 | 916,00 | 89,83 | 42,00 | clean |
| 59,00 | 21,40 | 16,00 | 187,00 | 916,00 | 190,00 | 42,00 | defective |
| 0,00 | 52,00 | 0,00 | 52,00 | 35,00 | 35,00 | 0,00 | clean |
| 0,00 | 0,00 | 0,00 | 0,00 | 916,00 | 916,00 | 0,00 | clean |
| 59,00 | 17,75 | 17,50 | 141,00 | 916,00 | 416,25 | 32,00 | clean |
| 7,00 | 3,67 | 5,00 | 26,00 | 659,00 | 264,33 | 5,00 | clean |
| 156,00 | 69,33 | 70,00 | 418,00 | 276,00 | 186,00 | 31,00 | clean |
| 0,00 | 1,00 | 0,00 | 1,00 | 351,00 | 351,00 | 0,00 | clean |
| 0,00 | 0,00 | 0,00 | 0,00 | 916,00 | 916,00 | 0,00 | clean |
| 0,00 | 1,00 | 0,00 | 1,00 | 351,00 | 351,00 | 0,00 | clean |
| ? | ? | ? | ? | ? | ? | ? | clean |
| 1,00 | 1,25 | 0,75 | 8,00 | 351,00 | 88,50 | 29,00 | defective |
| ? | ? | ? | ? | ? | ? | ? | clean |
| 5,00 | 10,57 | 2,71 | 93,00 | 916,00 | 244,29 | 48,00 | defective |
| 27,00 | 9,00 | 9,25 | 73,00 | 659,00 | 243,00 | 41,00 | clean |